A Monitoring-based Approach to Object-Oriented Real-Time Computing

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von Diplom-Informatiker Martin Gergeleit,
geboren am 23. September 1965 in Neuwied

Gutachter:
Prof. Dr. Edgar Nett
Prof. Dr. Wolfgang Schröder-Preikschat
Prof. Dr. Jörg Kaiser

Magdeburg, den 20. Dezember 2001

Table of Contents

List of Figures

1 Introduction

Future real-time applications will become large and physically distributed among different sites. They will have to work in environments that are so complex that it is not possible to specify all possible states and conditions in detail. At the same time, the time-to-market from the first idea to the working product decreases, especially as real-time technology is no longer restricted to long-living embedded applications, like in aircrafts or defense systems, but also penetrates business and customer applications. Many of today's Internet-applications have an increasing demand for real-time behavior. This starts from hard real-time telemetric-applications and critical trading applications, like stock trading, includes all kinds of multimedia-applications and ranges to multi-player games.

1.1 Motivation

An important issue, that all these applications have in common, is that they do not act in a closed environment. They act in environments that are usually not controlled by a single principal, like one company that runs an application-server or one telecom carrier that manages the whole network. Large scale distributed real-time applications (e.g. over the Internet) usually have to use resources from different entities. This immediately leads to a need for agreed common interfaces, i.e. standards that allow for interaction in a heterogeneous environment. Open standards have driven the expansion of the Internet and today it is commonly agreed that only applications that use and provide standard interfaces have the potential to become successful. Today, even closed applications (e.g. on the Intranet) are usually designed and implemented using these standards. However, these standards mainly have focused on functional behavior. Specification of timing behavior was initially out of scope. This is true for the major Internet protocol standards, like e.g. TCP/IP, IEEE 802.x, or HTML, as well as for commonly used interface description languages like CORBA or DCOM IDL. Also all common implementation languages (ranging from C/C++, Java to SQL) do not provide means to express timing behavior. What is needed are open standards that include timing and cope with the heterogeneity.

Object-Orientation

The benefits of object-orientation in system development and for handling heterogeneity are well accepted from the software engineering point of view. Today nearly all new IT systems emerge from an object-oriented analysis and modeling phase, their implementations are based on object-oriented middleware (like DCOM or CORBA), they are written in an object-oriented language (like C++ or Java), and their APIs are offered in abstractions of classes and objects. The desired properties of object-orientation design and programming are extensibility, reusability, understandability, robustness, portability, and efficiency achieved by

modularity, implementation hiding, and inheritance [Boo91]. It is common believe that these benefits also apply to real-time applications. For many real-time applications, the object-oriented approach (at least for modeling purposes) seems to be quite natural, especially for control applications, which have to handle real-world objects.

Timing Requirements

Only recently, extensions and new standards that can handle timing requirements or, more globally, QoS (Quality of Service) parameters have been proposed and introduced [OMG99b, OMG01]. QoS typically denotes a combination of many non-functional properties, including especially timing and fault-tolerance. While the use of QoS guarantees is common in the telecom world (where handling of synchronous and thus time-critical data is the core business), it is still a current topic of research in distributed computing. Especially the use of dynamic QoS guarantees is the challenge. While communication lines are usually leased with some static QoS assurances, it is highly inefficient and often infeasible to provide static guarantees also in a network of interacting services due to the number of involved components. Instead dynamic guarantees are negotiated and the resulting QoS is dependent on the actual requirements and the currently available resources.

Problem Exposition

A straightforward approach that combines the benefits of object-orientation and dynamic QoS guarantees seems to be a promising approach to tackle the problems of future real-time applications. A system as depicted in Figure 1.1 seems to be the ideal solution for the coexistence and interoperability of real-time and non-real-time applications in one common infrastructure. All components interact via a common object-oriented infrastructure and if this infrastructure manages also dynamic QoS guarantees, all problems can be solved.

However, such an architecture can only be successful in very isolated problem domains. For most applications and especially all Internet-based services this scenario is too simplistic. Even if all technical problems were solved, it relies heavily on the ability and, often even more important, the willingness of all involved entities to negotiate and implement the required QoS guarantees. Even with a complete replacement of the network, the operating system, and the middleware components with new, QoS aware versions, legacy application on top will still not be real-time capable. They have to be redesigned, often from scratch. With the large base of installed services, the migration process would require incredible investments, as the complete existing IT-infrastructure has to be replaced or at least updated. This leads to the situation, where real-time applications wanting to interact with the rest of the world will have to deal with non-QoS aware services for a very long time.

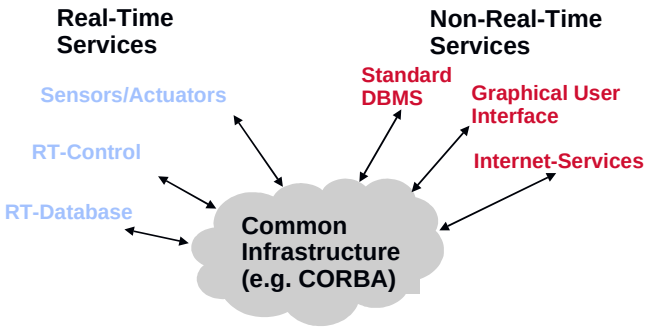


Figure 1.1: The simplistic view of a heterogeneous distributes real-time system

Another problem with this approach is that object-orientation itself imposes a major problem for QoS aware systems. While implementation hiding isolates module implementations and eases their integration, it ignores the fact that for negotiating timing and other resource-related guarantees detailed knowledge about the implementation is required. Implementation does matter! To that extend object-orientation and real-time computing are even contradicting. These observations were the starting points of this work: alternatives to the unrealistic scenario of the whole heterogeneous IT-environment being one big object-oriented real-time system had to be explored.

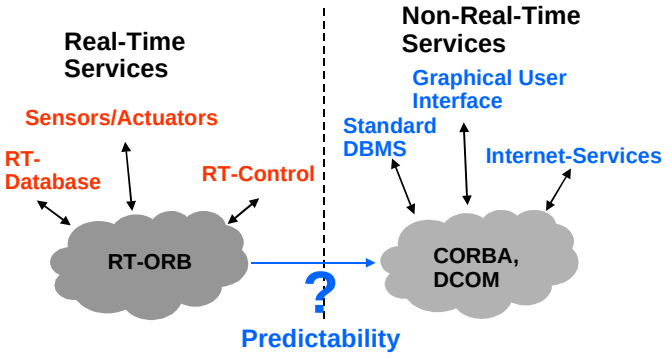


Figure 1.2: The proposed partition in distributed real-time systems

The idea is to preserve object-orientation as the basic paradigm for system construction and to undermine implementation hiding only where possible and where required. Clearly, it is required for objects that operate under strict timing constraints. Here, predictability is the most the important property. All other desirable features have to be reconsidered under this premise. On the other hand implementation hiding cannot be weakened for most of the external non-real-time objects. Either their code is simply not available or too complex to be ana-

lyzable in terms of resource requirements, or the administering entity will not grant access to it. This idea leads to an approach as depicted in Figure 1.2 where the applications are separated in two partitions, a real-time domain and a non-real-time domain. While the real-time domain itself is likely to be a distributed system based on a real-time-capable object-oriented infrastructure, it still has to interact with services that are running on top of standard middleware (like CORBA or DCOM) that does not handle timing requirements at all. The problem with this partitioning occurs as soon as an invocation crosses the boundary from the real-time into the non-real-time domain. Without additional measures, the real-time property of a computation is lost as soon as one of its components fails to meet the timing requirements. Thus, the fundamental question is: how to run real-time applications in a predictable manner in such a heterogeneous environment?

1.2 Approach and Outline

The first idea for handling unpredictably in a time-critical environment is the use of timeouts. Whenever the real-time system has to initiate an activity with an unknown time to completion, e.g. the invocation of a non-real-time component, it sets a timeout. If this timeout expires before the activity terminates, it executes some kind of exception handling. This mechanism ensures that the real-time system can detect the potential violation of timing bounds and react accordingly. However, there are two fundamental problems with timeouts:

1. *What to do if the timeout expires?* When the timeout expires there is usually not much time left to react. Some basic recovery and error reporting can be done in most cases, but how to ensure the computational progress? If there were an alternative algorithm (with guaranteed execution time) of achieving the same result as from the abandoned activity, it would have been applied directly. Using old results from prior executions or approximations can be an option, but only to a certain extent. If too many successive timeouts happen, the use of these increasingly imprecise values will lead to intolerable deviations in the final result. If this is not the case, there is obviously no need for the exact (but non-real-time) computation and the applications is over-specified.

2. *How to choose realistic timeout-values?* The worst situation that can happen is that a timeout is always a little bit too short. There is no computational progress but a lot of valuable resources are spent on computations that are abandoned every time shortly before their successful termination. On the other hand, if timeouts are chosen too long, the overall maximum execution time becomes greater than necessary. This results in a decreased throughput and thus a sub-optimal efficiency of the real-time system. The situation becomes even more complicated if the timing changes dynamically. Timeouts

that were adequate before may become too long or, even worse too short as the load characteristics in the non-real-time partition changes.

Both problems lead to the observation that there is an urged need for a precise estimation of the time required to execute certain activities and thus of the appropriate timeout values. The monitoring-based approach presented in this thesis tackles the problem by observing the timing-behavior of the system components. When guarantees are not available it enables the real-time scheduler to make decisions based at least on their *expected* timing. With this knowledge a fault-tolerant scheduler can determine realistic timeout-values, depending on the current system state, and plan online for alternative actions based on up-to-date statistical information.

The remainder of the thesis is organized as follows: In the remainder of the introduction, section 1.3, an overview of the existing approaches to object-orientation and real-time at the different architectural levels is given. Then chapter 2 describes time-awareness in distributed object-oriented real-time systems. Time-awareness reveals timing-related information of object-oriented systems from the non-real-time domain. The presented concepts, mechanisms, and implementations for monitoring these systems at the different levels of abstraction are the premises for chapter 3, where the main contribution, the monitoring-based approach is described. In this approach the gathered timing information from the running system is used to support efficient scheduling and intelligent timeout handling of not (totally) predictable objects. An implementation architecture is described and performance figures that prove the feasibility of the approach are presented. Related work on the specific topics of these chapters will be discussed in the concrete contexts. At the end of each of the chapters case studies for the application of the introduced mechanisms are given. A summary and references conclude the work.

1.3 Object-Orientation and Real-Time

As argued before, the traditional object model is insufficient in the context of real-time systems. Here a completely new aspect has to be added to the object concept, namely time. It has to be investigated how to annotate the functional specification of types with timing constraints and how to guarantee and implement these timing specifications. Also other concepts that were already included in the traditional object model have to be reviewed in the context of a real-time system, due to the difficulties to obtain deterministic timing behavior. These concepts include inheritance, dynamic binding, dynamic memory allocation, concurrency, and synchronization. A lot of research has been undertaken in order to resolve the inherent contradiction between object-orientation and real-time. In the following subsections several different approaches to real-time objects will be reviewed.

1.3.1 Object-Oriented Languages for Real-Time

The first attempts to enhance an object-oriented language, namely C++, to become real-time capable were the presentations of FLEX [Lin88, Ken94] and RTC++ [Ish90]. These languages contain special constructs for defining deadlines, periods, and synchronization conditions. Some of these constructs can be mapped to scheduler-level abstractions, like task-deadlines and priorities. The expressiveness of these language add-ons is limited (e.g. RTC++ provides no language constructs to define an aperiodic real-time task) and some timing constraints may be still implicit in the code. Both languages, RTC++ and FLEX, provide the basic information for a schedulability analysis, but they do not contain provisions for a static worst-case execution time analysis. Thus, timing violations still may happen. RTC++ contains an exception-handling mechanism that is activated *after* such a violation has happened. Flex supports computations with adjustable execution times by allowing them to return imprecise results [Liu94]. In addition, the runtime system can choose a version of a function based on performance constraints; this is called performance polymorphism.

Real-Time Specification for Java

A more recent approach to an object-oriented language for real-time is based on the Java language. Following the recommendation form the *National Institute of Standards and Technology* (NIST) the *Real Time for Java Expert Group* (RTJEG), a group of representatives from 21 organizations in industry, academia, and government, proposed a *Real-Time Specification for Java* (RTSJ) for real-time extensions to the Java language [Bol00]. The main underlying design principles are:

- *Compatibility*: the RTSJ shall not include specifications that restrict its use to particular Java environments. It shall not prevent existing, properly written, non-real-time Java programs from executing on implementations of the RTSJ and the Java idea of "Write once, run anywhere" should be preserved.
- *Predictable execution*: the RTSJ shall hold predictable execution as first priority in all tradeoffs.
- *No syntactic extension*: the RTSJ shall not introduce new keywords or make other syntactic extensions to the Java language.
- *Current practice versus advanced features*: The RTSJ should address current real-time system practice as well as allow for the incorporation of more advanced features in the future.

Unlike most Java specifications that merely define new APIs, the real-time specification provides modifications to the Java language specification and the Java Virtual Machine (*JVM*) specification, as well as new APIs. This means, Real-Time Java applications will need a special JVM on which to execute, but

could use many of the features of the standard Java programming model. The RTJEG identified basically five areas for modification:

- *Scheduling*: the RTSJ allows the programmatic assignment of parameters appropriate for the underlying scheduling mechanism in use in a given real-time system, as well as providing methods for the creation, management, admittance, and termination of real-time Java threads. The RTSJ base scheduling mechanism is preemptive priority-based, FIFO within priority, with at least 28 unique priority levels. However, the RTSJ is open for future extension to load other schedulers as well.
- *Memory management*: the RTSJ defines a memory allocation and reclamation specification that is independent of any particular garbage collection algorithm and lets the program precisely characterize the garbage collection algorithm's effect on the execution time, preemption, and dispatching of real-time Java threads. The RTSJ defines new types of memory areas, *ImmortalMemory* and *ScopedMemory* that allow the creation of Java objects but do not cause the threads that employ them to incur delays because of the execution of the GC algorithm.
- *Synchronization*: the RTSJ defines that the semantics of the current Java keyword "synchronized" has to be enhanced. Instead of pure mutual exclusion priority inheritance is provided by default.
- *Asynchronous event handling*: the RTSJ generalizes the Java language's notion of asynchronous event handling. The *AsyncEventHandler* class is extended to run as real-time thread when the event is triggered.
- *Asynchronous transfer of control*: the RTSJ specifies that methods that allow for being interrupted (receive an exception) by another thread at any time. This is an extension to plain Java, where this could happen only in certain blocking calls. This mechanism can also be used for terminating a tread by an external event.

A Reference implementation of the RTSJ is currently under development.

Discussion

While object-oriented languages like FLEX and RTC++ try to exploit the features of object-orientation for a simple and flexible programming of real-time systems, their applicability is rather limited. They provide the expressiveness to define the typical tasks of a real-time system, but both do not support a full schedulability analysis as required by a hard real-time system. Both languages require their own underlying runtime-system and they are closed in a sense that they were not designed for interfacing with external objects.

Things are a little bit different for the RTSJ. It provides a perfect basis for the interoperability of real-time and non-real-time objects and with its powerful APIs for scheduling and thread-management it will surely provide an interesting

platform also for the concepts presented in chapter 3 below. However, RTSJ does not define "real-time objects" but it uses an object-oriented language to program a real-time system. It is more an operating system extension than a language. Timing specifications are not part of an object's interface but they are implicit in the code. Whether or not a schedulability analysis is supported depends on the actual implementation of the runtime-system, but as the RTSJ does not impose any restrictions on the Java language, this is an inherently hard problem.

All languages do not address distribution at all. In the case of Java this imposes additional problems, as other extensions to the Java-language already have a clear orientation towards distribution (e.g. Java RMI and the Java CORBA-binding). A combination of these different directions in Java evolution is still future work.

1.3.2 Object-Models for Distributed Real-Time

In the research community a number of extended object-models were proposed that try to provide both, distribution and predictability. Two of them will be presented in more detail.

TMO

The TMO (*Time-triggered Message-triggered Object*) [Sho99] scheme is a good example for this approach. It has been first published in the early 1990's, previously named RTO.k. The TMO structuring is intended to support the design of all types of components including heterogeneous systems with real-time and non-real-time objects within one general structure. The basic TMO structure is depicted in Figure 1.3. The significant extensions of TMO compared to the traditional object model are:

1. *Distribution:* A TMO is a distributed computing component. TMOs are distributed over multiple nodes and interact via remote method calls. To maximize the concurrency in execution of client methods and server methods, client methods are allowed to make non-blocking types of service requests to server methods.

2. *Time-triggered and method-triggered methods:* The TMO may contain two types of methods, time-triggered (*TT*-) methods (also called spontaneous methods, *SpM*s) and the conventional message-triggered (*MT*-) methods (also called service methods, *SvM*s). The TT-method executions are triggered upon reaching of the real-time clock at specific values determined at the design time. Each TT-method is associated with an autonomous activation condition (*AAC*) that specifies the times at which the associated method should be activated. The MT-method executions are triggered by service request messages from clients.

3. *Concurrency constraints:* A fundamental concurrency constraint prevents potential conflicts between TT-methods and MT-methods and reduces the designer's efforts in guaranteeing timely service capabilities of TMOs. Activation of an MT-method triggered by a message from an external client is allowed only, when potentially conflicting executions are not in place. An MT-method is allowed to execute only if no TT-method that accesses the same set of resources within the object will overlap with the execution time-window of this MT-method.

4. *Guaranteed completion time and deadline:* As in other RT object models, the TMO incorporates deadlines and it does in the most general form. Basically, for output actions and method completions of a TMO, the designer guarantees and advertises execution time-windows bounded by start times and completion times. Triggering times for TT-methods must be fully specified as constants during the design time. It is also possible to specify so-called *candidate* triggering times in contrast to the actual triggering times. A subset of the candidate triggering times may be dynamically chosen for actual triggering. Such a dynamic selection occurs when an MT-method within the same TMO object requests future executions of a specific TT-method.

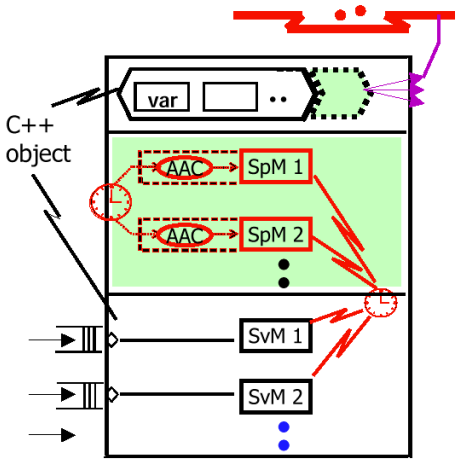


Figure 1.3: Structure of a TMO object

TMOs interact via invocations of service methods in server objects from client objects. The caller may be a TT- or an MT-method in the client object. It is up to the designer of each TMO to provide a guarantee of timely service capabilities of the object. The designer does so by indicating the guaranteed execution time-window for every output produced by each MT-method as well as by each TT-

method executed on requests from the and the guaranteed completion time for the in the specification of the MT-method. These specifications are advertised to the designers of potential client objects. A middleware that supports the TMO model and that is based on the standard CORBA API has been implemented on Windows NT.

Real-Time CORBA

The Real-Time CORBA (RT CORBA) specification [OMG99b, Sch00] adopts a similar approach. It extends the CORBA standard by interfaces and QoS policies that allow applications to configure and control the various resources:

1. *Processor resources*: RT CORBA defines thread pools that allow to control the mapping of multiple treads to certain interfaces. It also introduces activities as a kind of distributed threads (as described later in more detail) and global priorities that enforce a fixed priority scheduling throughout the distributed system. Priority inheritance and priority ceiling protocols are provided by new intra-process mutexes and a global scheduling service hides the platform-specific details of low-level resource management under a common API.

2. *Communication resources*: RT CORBA allows to specify required protocol properties explicitly and and it adds APIs to perform explicit binding of communication streams to certain interfaces.

3. *Memory resources*: Management of memory resources is made explicit by RT CORBA as it allows to manipulate the buffering policies in queues and to limit the size of thread pools.

RT CORBA has been implemented first by the TAO ORB [Sch97], but other vendors are following soon after.

Discussion

The TMO model provides a rather strict framework for the development of distributed hard real-time applications. Because of its restrictions it simplifies schedulability analysis. It extends the object-oriented model by adding timing specifications to the interface. However, even if TMO now supports CORBA interfaces and can interact with external objects, it still can not preserve predictability when leaving the real-time system that consists of a homogenous network of TMOs.

RT CORBA can be seen more as an abstract distributed operating system than a new object model. It encapsulates many features of operating systems and maps them to a heterogeneous, distributed environment. To that extend it supports heterogeneity of platforms, but it also does not foresee any mechanism for integrating non-real-time services, other than the plain possibility to call them via the standard CORBA APIs.

1.3.3 Object-Oriented Modeling for Real-Time

A different approach to object-orientation than object-oriented languages or middleware layers has evolved over the last ten years from the community that developed and promoted general-purpose object-oriented design and modeling methodologies. Two of them will be presented in more detail: SIMOO-RT, because it has been enhanced in a joint work with the Federal University of Rio Grande do Sul, Brazil with the monitoring components described below in chapter 2 [Bec99] and Real-Time UML, because UML is now the well-accepted standard for object-oriented design.

SIMOO-RT

The SIMOO-RT environment [Bec00] is a modeling, design, and simulation framework. The extensions to the standard features of object-oriented modeling tools comprise the explicit representation of timing requirements like deadlines, timeouts and periodic operations.

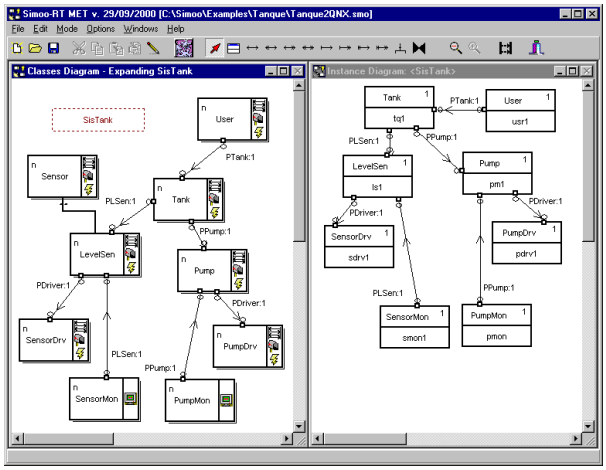


Figure 1.4: SIMOO-RT Model Editing Tool (MET)

The first step in the development process with SIMOO-RT is the definition of an object-oriented model for the problem under analysis. The *Model Editing Tool* (MET) provides support for the construction of two different diagrams: the class diagram that depicts important problem domain concepts and their relationships, and the instance diagram that represents the specific elements that take part in a specific application. Figure 1.4 depicts a screenshot of the MET, where the most left part represents the class diagram and the right one represents the instance diagram.

For modeling the internal object behavior, the environment encourages the use of state-transition diagrams. Incoming messages are associated to actions that ob-

jects have to convert in reaction. These actions can be executed either during the state transition as well as while the object remain in a given state. Temporal constraints can be imposed to the model by specifying *cyclic operations* and *deadlines*. Also, a pre-defined *timeout* exception handler can be specified. This operation is only activated when the given operation doesn't accomplish its deadline. In the SIMOO-RT environment timing properties are attached to the classes.

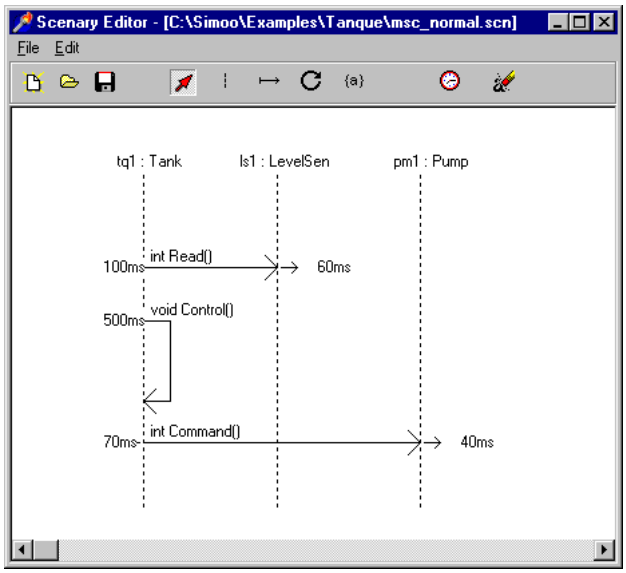


Figure 1.5: A SIMOO-RT Message-Sequence Diagram

For specifying the model global interaction SIMOO-RT allows users to state the whole set of a system's functionality by means of UML use-cases. Furthermore, each use-case can be detailed into a Message-Sequence Diagram (MSD) or into a data-flow diagram (DFD). Through the MSDs, the user can establish the object interactions, as well as timing-marks that represent the overall systems timing behavior. An example of a MSD is depicted in Figure 1.5. Once a design is ready for execution it can be tested in an internal simulator or it can be send to an automatic code generator for executable code. The resulting implementation than can be executed in a distributed environment using the QNX real-time operating system as underlying platform.

Real-Time UML

The UML (*Unified Modeling Language*), an OMG standard, is a language for specifying, visualizing, constructing, and documenting software systems [Boo99, OMG99]. UML fuses the concepts of Booch, OMT, and OOSE (all three older OO modeling techniques). The UML focuses on a standard modeling language,

not a standard process. Therefore, the efforts concentrated first on a common meta-model (which unifies semantics) and second on a common notation (which provides a human rendering of these semantics). The UML authors promote a development process that is use-case driven, architecture centric, iterative and incremental. The UML is intended to be a visual modeling language, not a visual programming language, in the sense of having all the necessary visual and semantic support to replace programming languages. However, the UML has a tight mapping to a family of object-oriented languages and many tools support C++ as their primary language.

Currently the *UML Profile for Schedulability, Performance, and Time* is being in the OMG standardization process [OMG01]. It proposes "standard paradigms of use for modeling of time-, schedulability-, and performance-related aspects of real-time systems that enables the construction of models that could be used to make quantitative predictions regarding these characteristics". It tries to unify the various different approaches that have evolved for modeling real-time application in UML. These are namely the approaches from Douglass [Dou98] that uses mainly the existing predefined UML diagrams, and the work from Selic that uses extensively the option of UML to define stereotypes (a kind of visual macros in diagrams) to impose a notation similar to ROOM (*Real-Time Object-Oriented Modeling*) [Sel94] his prior modeling environment onto UML.

The abstractions and diagrams specified by UML are similar to those of SIMOO-RT, while partially more elaborated. The fundamental notation for expressing object-timing relations is again the message sequence diagram. In RT-UML it has been extended and it can be annotated with expressions in a constraints language that allow to define exactly the relation between the occurrence of events and messages. The various tools from the different vendors that promote UML for real-time can automatically generate code that runs on a number of real-time operating systems.

Discussion

While the expressiveness of UML and similar modeling tools is powerful and with UML it is possible to describe all kinds of distributed object-oriented computing in a standardized manner, it also cannot solve all problems in the real-time domain. The available methodologies (as manifested e.g. in the products from Rational and I-Logix) are a step towards a more formal, partially automated development process of distributed real-time control systems. However, they are far away from providing a proof for the correctness of the synthesized code. Still it is possible that the specification of the system is correct and complete and the synthesized code is simply not able to fulfill the timing requirements (due to errors in the design or insufficient resources of the executing system).

1.3.4 Summary

Basically, the different approaches to real-time objects can be clustered in two categories: the *operational* and the *specificational* approaches.

The real-time languages (especially RTSJ) and RT-CORBA are focused on providing access to and control over the resources of the underlying machine, not on an extended object-model. In these *operational* environments it is the believe that the programmers will be clever enough to use the enhanced control correctly to achieve what they want and deploy the created objects in the right manner. Implementation hiding is not weakened by the object-model but by the people that use the real-time programming environment and document their created classes. Schedulability analysis is not an issue here.

The *specificational* approach lead by UML allows to define exactly the timing (and the functional) behavior of the implementation. This is indeed a step towards a special object-model for real-time. However, the modeling and design environments do not care to much about how to achieve this correct behavior. The high level abstractions they use during design need to be mapped without losses down to the real implementation. Especially in a heterogeneous environment, where not all components can be simply synthesized from a specification, this clean approach reaches its limits.

The TMO-approach can be seen in the middle: while it deals with low-level abstractions, it allows to specify timing at the interface. With its strict design rules it supports schedulability analysis. Another combination of both approaches is surely on the way, e.g. a Real-Time UML design environment with RTSJ or RT-CORBA back-end (all (future) OMG standards). This will probably lead soon to a satisfying result for objects in the real-time domain. However, it has to be stated that none of these approaches has a convincing solution, how to integrate the rest of the IT-world, like Internet-based services, into real-time computations. The question mark on predictability from Figure 1.2 is still in place.

2 Time-Aware Systems

The first step in a monitoring-based approach to object-oriented real-time computing is a *time-aware* system. A system is time-aware if it is able to get information about its own timing, e.g. to monitor execution times or to check whether the actual timing conforms to the specification. Even a non-real-time system, i.e. a system that provides no time-related guarantees, can be time-aware. Time-awareness enables a system to react on timing-related problems and to adapt to an actual, but previously unknown, timing behavior in a concrete application environment. This in turn is the basis for active QoS-management and a key to handle the inherent problems of heterogeneous object-oriented real-time systems that were outlined in the introduction. If it is not possible to provide guarantees when non-real-time object are invoked from the real-time domain, it is at least helpful to get an idea of what is going on and what might be the problem.

However, the typical time-awareness of today's systems is poor and limited to an API that allows an application to read a global clock or a thread-relative timer. Any system that implements ad-hoc time-awareness based on these primitives at application-level has to address the same problems again and again. As soon as time-awareness is accepted as a requirement of a class of applications, it is a perfect candidate for becoming a system-level service. Moreover, there is already a well-engineered category of system-level tools that handles exactly these problems in a generic and efficient way, namely the *monitors*.

This chapter explains how the existing approaches for monitoring can be adapted and combined to provide the required system-level time-awareness service for object-oriented systems. It first introduces the common terminology of monitoring systems and then reviews the state-of-art. Then, it discusses in three subsections the special requirements for the monitoring of object-oriented systems at the different architectural levels, namely the operating system level, the middleware level, and the programming language level. Finally, it presents as a case study a monitoring tool that integrates the monitoring at these different levels in one tool.

2.1 Monitoring

As defined in [Tsa96], monitoring a system means to collect runtime information about the *system under test* that cannot be obtained by static analysis, i.e. by only analyzing the program code. A *monitor* is a system used to monitor a system under test's execution. The system's behavior, including the behavior of the application program and the operating system, can be described as a series of *events*. These events are the visible changes of the system under test's state, e.g. process creation or termination, sending or receiving a message, or context switches. Usually, events are grouped by categories. The different events of one

category are distinguished by a number of parameters (e.g. "process creation" is the event category and each event of this category is distinguished in the parameter of the actual process number). Depending on the focus of interest or the level of detail only a small subset of all possible events is usually observed by a monitor. Those events that are monitored are called the *events of interest*. *Sensors* are used to record the events of interest. An *instrumented system* is a system with added sensors. When an instrumented system is executed, *event traces* are produced.

The required activities for monitoring are the *instrumentation* of the system under test (specification of events by inserting sensors into target programs), *event detection* (identifying the event occurrence generated by die execution of the instrumentation code), and *event processing* (time stamping and storing the parameters collected at the event occurrence in an event trace).

Monitoring intrusion refers to any attempt to record system execution by using the computing resource of the monitored target system. The problem with monitoring intrusion is that it causes *interference*, i.e. a perturbation of the execution of the system under and thus a difference in the timing and in a system with concurrent threads of control possibly even in the functional behavior.

2.1.1 Types of Monitors

Monitors can be classified according to their method for observing the system under test's state (event-based or sampling), to their implementation (hardware or software) and to their abstraction level (system or application).

Event-based or Sampling Monitor

An alternative approach to *event-based monitoring* as defined in the previous section is *sampling*. Sampling is a time-based technique, where a small part of the system state is captured and recorded with a certain *sampling frequency*. In contrast to e.g. sampling of an electrical signal, where oversampling is used, the sampling frequency of a (software) monitoring system is usually much smaller than the maximum frequency of state changes. Thus, sampling is typically used, if the desired result of the monitoring activity can be obtained by a statistical analysis. This means, sampling is not appropriate for short monitoring intervals as it relies on a large number of samples for achieving sufficient coverage and confidence. Also, as it does not capture information on all relevant state changes nor on their exact sequence, sampling is hardly applicable to obtain information on problems that happen only a few times during the monitored execution. However, this is exactly the case for many timing-related problems in real-time systems. Thus, monitoring in the real-time context is typically done with event-based monitors.

Hardware or Software Monitor

Event detection and processing can be performed in a number of different ways, each of them causing a different amount of interference with the system under test. The more dedicated resources are available for measurement data processing, the less interference with the system under test can be expected. In principle, event detection and processing can be done with software, hardware, or a combination of hardware and software. The difference between *hardware* and *software monitoring* is that the hardware approach separates the monitoring task from the target system's workload, whereas the software approach adds to the target system's workload. A *hybrid monitor* refers to a monitor that uses a combination of hardware and software.

The optimal solution for monitoring with respect to interference is a hardware monitor that is able to detect events without affecting the activity of the system under test. This is accomplished usually by passively monitoring the target processor's signal lines such as the data, address, and control buses. However, as argued in [Hab89] hardware monitors have reached their frontiers due to the extensive use of memory management units and on-chip caches in today's computer systems. Most of the state changes of the software isn't reflected any more by signals on externally accessible signal lines, but they result only in some on-chip operations. It is out of scope for the addressed problem domain to try to get access to these on-chip signals for monitoring purposes. But even if this were possible, difficulties arise in identifying events in a problem-oriented manner. Often, a lot of different logical events are mapped to the same measurable physical event (e.g. access to the same memory location from different contexts). If the internal state of the system context has to be known to detect an event, a type of software component is needed which is inserted into the code of the system under test at locations corresponding to the events of interest.

This leads to an hybrid approach to overcome these problems. Here sensors are divided into an internal software part and an external hardware part which runs on resources dedicated solely to the monitor. The internal part consists of additional statements inserted into the code of the system under test in order to detect events of interest. Upon activation the event handling routine extracts the parameters associated with that event and passes them to the external sensor part for further event processing. While this approach combines the flexibility of software sensors with the low interference of a hardware monitor, it suffers from the required special purpose hardware. Usually, a hybrid monitor for a distributed system consists at least of a dedicated global clock, an event-processing unit per node of the distributed system with a high-speed interface to the system under test and some sort of network for propagating the events traces to a central monitoring console. This means, such a monitor is a complex distributed system in itself. This additional effort can be justified for a number of experiments in the testing lab, but it will not be tolerated by a customer as a permanent part of the application if it increases the overall costs significantly.

As this thesis argues that monitoring should be an integral part of the runtime system for enabling time-awareness, for most application domains, pure software monitoring is the only viable solution. Here the monitoring system consists solely of additional software inserted into the target system code. Event detection is accomplished by executing the inserted sensors, parameters pertaining to events of interest are recorded and stored in the memory of the system under test, and event traces are transferred using the same network as the application. Thus the software monitor shares the computing resources with the monitored target system. As a result instrumented programs have an execution speed penalty. Without the use of additional resources, the dilemma of finding a balance between minimal interference and recording sufficient information always exists. Limiting instrumentation provides inadequate measurement detail, but excessive instrumentation will perturb the measured system to an unacceptable degree. In order to tackle these problems, techniques for reducing the interference while retaining sufficient information have been developed. Interference can be reduced by an optimized instrumentation mechanism, by modifying the monitored target programs, or by selectively switching on and off the events of interest depending on the current status of the system under test and the monitor. Also, in a kind of post-processing the event trace can be adjusted to reduce the effect of interference.

System or Application Level Monitor

Depending on the motivation for monitoring execution behavior can be monitored at system level and/or application level. At system level, activities and data structures visible to the operating system kernel and all generic components of the system (like e.g. communication system or middleware) are monitored. At the application level, activities and data structures visible to the user processes are monitored. Those visible at system level include process state transitions, external interrupts, system calls and interprocess communication. Those visible to the user processes include function/method calls and returns, and variable value changes. Some of the activities and data structures, such as system calls, are visible to both the system and the application level, and others are visible only to the kernel, like process state transitions and interrupts. Finally, others are visible only to the application level, like function/procedure calls and returns and variable value changes.

To monitor at the system level, the kernel and other runtime services can be instrumented for the events of interest. As this instrumentation is inserted into generic components it can be applied once and reused even if the application running on the system changes. This instrumentation typically requires only a very limited number of sensors. However, as the execution frequency of this sensor code is often quite high (e.g. hundreds to thousands of interrupts per second) careful optimization is needed in order to minimize interference in case of temporal instrumentation or inefficiency in case of a permanent instrumentation.

Thus, system level instrumentation is usually applied manually and there is nothing wrong with that.

In contrast, application level instrumentation tends usually to be quite big. As user-level function-invocations are usually not routed via common code sequences, each and every function has to be instrumented separately. While the required techniques for event detection and parameter collection are always the same, applying this instrumentation manually is still inconvenient, error-prone, and time-consuming. This opens a perfect scenario for automating this process.

2.1.2 Related Work on Monitoring

Over the last two decades monitoring tools have shown their value for debugging and performance evaluation of complex software systems.

Sampling Monitors

The probably most used tools for program-monitoring are the Unix tools “prof” and “gprof” [Gra82] and their deviates for nearly an system platform or language. They use a sampling approach and their monitoring model is a call-graph that presents a breakdown of the total execution time per (C-)function. While similar tools are also available for object-oriented languages like C++ and Java, their results suffer from their pure functional model. The information provided reflects only a static view on methods (a class-view) and object contexts are ignored. Many programs have been tuned during the last decades using these tools and their model is well suited for non-real-time programs that run on a single CPU. But for distributed real-time applications these tools have reached their limits, as they don't address distribution and concurrency. In the following a number of event-based software monitoring systems that explicitly address real-time issues will be summarizes.

Event-based Monitors

In 1992 the distributed measurement system JEWEL [Lan92] was presented. JEWEL consists of a generic set of flexible components and is not limited to a specific application domain. The JEWEL components are a configurable graphical presentation system for online visualization of the behavior of the system under test, a central interactive experiment control system, and a modular, distributed event processing system. JEWEL was designed and implemented to provide results of high precision. This goal was achieved primarily by ensuring that interference between JEWEL and the system under test is kept low. A clear separation of the sensor functionality made it possible to take full advantage of the properties of the different hardware/software environments, e.g. by using dedicated resources to achieve low-interference. JEWEL allowed for the use of hybrid sensors as well as for a pure software monitoring approach. It has been adapted to a number of different target architectures and systems under test. An implementa-

tion for monitoring real-time application on embedded processor boards running VxWorks was based on additional dedicated monitoring processors which were attached to the system under test. The second implementation for observing the performance of the Mach 3.0 operating system has been primarily targeted towards multi-processor systems and was done completely in software.

While JEWEL implemented all architectural levels of a distributed monitoring system as described e.g. in the book of Jain [Jai91] or Tsai [Tsa96] and revealed the desired flexibility it suffered from two related basic problems. Firstly, it implemented no default monitoring or measurement model, i.e. it doesn't suggest the user any structure to start monitoring a system, like e.g. the 'prof' tool does by creating the annotated call-tree. Secondly, it also left all the burden of instrumenting the system under test completely to the user. As JEWEL only provided the infrastructure for monitoring it had no idea where to place the sensors and which parameters to collect.

System-Level Monitors

Miller, Macrander, and Sechrest [Mil86] described in 1986 a measurement tool for monitoring the execution performance of distributed programs running in BSD UNIX. A model of distributed computation and measurement is used to describe the activities of processes in terms of their internal and external events (corresponding to computation and communication). Based on this model, a monitor and a measurement tool was constructed by changing the kernel-level structures of BSD UNIX and adding some daemon processes to allow the monitor to observe distributed activities that cross machine boundaries. The measurement system consisted of four parts: the meters, the filter processes, a control process, and analysis routines. Each node has a meter in its kernel. Implementing the meter inside the kernel avoids context switching and thus reduces the degree of interference. The meter detects events and extracts the event parameters from the operating system's data structures. The meter detects interprocess communication events by intercepting system calls made by the monitored processes. From the meter the events are sent to a possibly remote filter process. The filter process selects and reduces the received event data according to configurable selection rules. The filtered event traces then can be written to a file on be analyzed online.

Unlike the later JEWEL tool, this monitor was tailored towards a specific target architecture. This allowed for providing a generic instrumentation inside the target operating system. Without further instrumentation a user of the monitor receives an event trace of all relevant system level events. However, the main tasks that remain were to relate these events to user-level (i.e. programming language-level) activities, to reduce the huge the amount of data by appropriate filtering and to present the data in an understandable manner.

The systems described so far tried to minimize the interference. Dodd and Ravishankar from the Real-Time Computing Laboratory at the University of Michi-

gan have chosen a different approach. They tried to predict it. In [Dod92] they proposed the monitoring system HMON for the real-time system HARTS. HMON is able to provide consistent monitoring and deterministic replay by predicting the overhead caused by monitoring. HMON assumes that the complete system is predictable because the monitoring code is part of the target system. However, the design of the HMON monitor is closely dependent on the hardware architecture of HARTS. The nodes of HARTS are connected via a hexagonal mesh interconnection network. Each node is a tightly coupled multiprocessor system and is directly connected to six neighbors. The nodes have up to three application processors, a network processor, an Ethernet processor, a system controller, and a monitoring processor. In addition an external workstation is used for logging the event traces. All parts of the monitor are implemented in software. However, the required amount of additional hardware resources is significant. Thus, it can hardly be categorized as a pure software monitor. The generic HMON instrumentation detects the following events: system calls, interprocess communication, interrupts, and application-specific events.

While HMON is able to visualize monitoring data, the main focus of the project was on determinism and on the ability to replay a distributed execution on the real system. This ability is tightly coupled to the specific system environment and cannot be transferred to a typical current real-time computing environment. In subsection 3.1.1 it is argued in more detail that there in an inherent tradeoff between today's high-performance hardware architectures and fully predictable behavior. Also, it is impossible to preserve the degree of predictability that is required by a monitor like HMON as soon as the system has to interact with other computers that are not part of the monitored domain.

Tokuda, Kotera, and Mercer proposed in 1988 a real-time monitor featuring the visualization of the internal behavior of a distributed real-time operating system ARTS [Tok88]. It consists of a real-time monitor/debugger to visualize the target systems scheduling decisions in quasi-real-time by Gantt-diagrams. Information is gathered by a software sensor, called an Event Tap, embedded into the real-time operating system kernel. To predict and reduce the monitoring interference, the monitor is a permanent part of the ARTS system so that scheduling always includes the overhead of monitoring.

The main contributions of the ARTS monitor were permanent instrumentation (like in HARTS/HMON) and online visualization of real-time scheduling. While permanent instrumentation is still sometimes considered too much overhead, the type of visualization is well accepted in the community for understanding the behavior of real-time systems. It has been adopted e.g. by the commercial tool WindView for the VxWorks real-time kernel.

Monitoring of the Middleware Layer

The systems presented so far, do not address standard middleware layers. Work on instrumenting and monitoring of object-oriented middleware has been done

by Rackl [Rac00] with MIMO (*MIddleware MOnitor*). The tool it targeted towards heterogeneous systems under test and special emphasize it put on the multi-layer approach. It provides a framework that conceptually allows to monitor the system at six different levels: application, (object-oriented) interfaces, middleware (distributed objects), language, operating system, and hardware. This approach is based on the same observation that lead to the monitoring tool MagicZoom described below in subsection 2.5: the fact that a consistent mapping between the monitoring information gathered at the different layers is of great importance for understanding the system. In MIMO special emphasis is put on the instrumentation of the middleware layer. In [Rac01] two concrete implementations for instrumenting CORBA and DCOM are presented, both differ from the approach presented in this thesis. The CORBA instrumentation is based on a special instrumented library that replaces the original CORBA library. The DCOM approach is based on a *universal delegator* [Bro99], a redirection of DCOM invocations. The presented solutions are capable of intercepting all required object-related calls. However, the major problem is performance. With an overhead of 40 to more than 100% the interference is enormous. This results from the fact that the monitoring component itself is implemented using CORBA. MIMO itself can be considered as a kind of generic distributed event propagation service. While this is a clean approach, it is surely not suitable for online monitoring of real-time systems. The envisaged applications for MIMO are therefore system management scenarios.

Summary

Concluding from the numerous different monitoring solutions it can be stated that the four main problems every distributed monitoring system is facing are:

1. how to limit and or at least predict the level of interference,
2. how to synchronize local clocks or to provide an additional event ordering mechanism,
3. how to minimize the effort of instrumentation while providing traces with a sufficient level of detail, and
4. how to transport, process, and present the event traces in a manner adequate to the problem domain.

As all these problems can be solved with different tradeoff considerations in mind, the potential design-space for an event-based monitor huge. There is no dominant standard available in this area and none of the so far mentioned monitors addresses all of the requirements of an object-oriented, standard-based real-time system. Therefore, the monitoring solutions presented in the remainder of this chapter are focused on this special class of systems.

2.1.3 Monitoring Object-Oriented Real-Time Systems

In the context of this thesis the system under test is the observed real-time system, i.e. the object-oriented, distributed system that executes the time-critical application. Its software consists of the application programs (written in some programming language), the object-oriented middleware, and the operating system. (As the operating system is the lowest software layer its monitoring also covers the influences from the underlying hardware layers.) Now it has to be considered, which parts of such a system have to be monitored in order to provide the required information for time-awareness? All components that contribute to the system's timing. As timing is a non-functional property that cannot be fixed to one abstraction layer of the system, consequently time-awareness requires monitoring at all system levels.

2.2 Instrumentation at the Operating System Level

The lowest software level that contributes to the timing of the real-time application is the operating system. As the operating system is finally responsible for the assignment of resources, including the CPU, it is often in the focus of interest when the actual results of a scheduling strategy adopted by an application have to be analyzed. The related abstractions provided by an operating system are processes, threads, interrupts, and synchronization objects, like e.g. messages or semaphores. As the examples of ARTS and WindView (see above) have shown, detailed event traces on the state changes of these objects are inevitable for an understanding of the system's scheduling behavior any monitoring system that claims support real-time computing should provide this information.

2.2.1 Issues in System-Level Instrumentation

When instrumenting the operating system and collecting events at this low level a number of important implementation-related issues and restrictions have to be obeyed:

1. **How to place a sensor to get aware of an event of interest?** As long as the source-code of the operating system kernel is available, as this is the case e.g. for Linux, RTLinux, and partially also for Windows CE the code can be analyzed and the sensors can be placed at the right locations. Then a new instrumented kernel can be build and this kernel can then be used for the system under test. For all systems that do not provide a build environment for the kernel this is not a viable solution. Here the existing kernel has to be modified to execute the additional sensor code. As long as the kernel provides the required hooks (i.e. debugging APIs) that allow for adding this code at runtime, this is also a simple job to do.

2. **How to obtain time-stamps with sufficient accuracy?** Typically, kernel-level events like context-switches and interrupts happen at rates higher than 1 event/ms. This means, the standard operating-system timer running with at most 1 kHz does not provide sufficient resolution for accurately time-stamping these events. At least a microsecond resolution is required. A microsecond-counter with only 32 bits wraps around every 71 minutes (4294,967296 seconds). Hence, any long-term measurement needs time-stamps with significantly more bits, typically 64.

3. **How to minimize intrusion?** As stated before, operating system level instrumentation can produce event-rates above 1 event/ms. With an execution time of about 10 µs for one sensor this can already result in an overall slow-down of more than 1%. While, this is a fundamental drawback of any software monitor and differences in the monitored case can be avoided by using permanent instrumentation, it is still the goal to minimized intrusion as far as possible. Careful coding of the sensors is required to accomplish that.

4. **Where to collect event-traces?** As operating system level instrumentation typically deals with very low-level events, the sensor code is usually executed either in an interrupt handler or in some locked kernel state. This means when the sensors write event data to a memory buffer, this buffer must be accessible from this execution level and under no circumstances the kernel can trap into a page-fault. A page-fault would introduce an arbitrary amount of intrusion and even worse, in a locked kernel state it would crash the machine. Therefore, the memory buffer has to reside in the same address-space and it must be pinned to physical memory. Other event-sources, e.g. in user-level code, may use different buffers. Events from different buffers can be merged into one linear event trace off-line using time-stamps from the local clock. However, as the need for minimized intrusion forces all buffers to be pinned to physical memory, it is an obvious idea to use just one buffer for all events. Given that writing one events is an atomic action, this buffer will then contain all events in chronological order.

Even if these problems are solved convincingly by a monitor, there is still the additional challenge of distribution. Local traces from the involved nodes have to be merged into one view of the system. This requires the transport of the possibly large amount of data and also, even more serious a timely synchronization. A general property of distributed systems is that there is no global clock. There is a lot of work done in the causal ordering of events using just local clock, but in the scope of monitoring this is of limited value. In order to allow for the timely correct observation of activities that span across a number of nodes, some kind of a global timer is required. If one is interested e.g. in the latency of a (one-way) message sent over the network, the only way to measure this time is to correlate the (global) time-stamps of the “message send” and the “message receive” event. One way to generate a global time-stamp is to use special monitoring hardware implementing a true synchronous timer that can be accessed by the sensors. But

even if there is no special global clock device available (e.g. a GPS-based clock) in the envisaged environment, traces can be synchronized a-posteriori. This can be done by identifying at least two events in each of the involved traces that have a known timing relation (given that the local clocks have an unknown offset and a constant drift) [Ger95].

2.2.2 Implementation for Windows NT

In order to discuss concrete solutions to these important issues in monitoring and instrumenting at the operating system level, Windows NT 4.0 has been chosen as a representative target system. The concepts presented with the JewelNT monitoring tool are portable and have been applied in a quite similar way also to Windows CE, VxWorks (TORNADO), RTLinux, and Mach 3.0 [Ger92]. From the technical point of view Windows NT can be considered as one of the most difficult targets for instrumentation, as it doesn't include predefined hooks for installing sensor code (like VxWorks) nor is it available in source code as the other mentioned targets.

The standard monitoring tools provided by NT are not sufficient for the desired application domain. Like the performance Monitor 'perfmon.exe' they rely on the performance counter API that is designed to provide average values but no information on individual events. More detailed traces of system events with accurate time-stamps are not provided. This means, no information is available e.g. on minimum and maximum duration of certain executions. Also, these tools are not designed for distributed computations. While they are able to access performance data of remote machine, they do not provide information on cross-context or even cross-machine activities (e.g. message latencies).

The JewelNT monitoring tool has been designed as a software-only, event-driven, distributed monitor for Windows NT. JewelNT allows analyzing the event traces on a remote machine with a graphical presentation interface [Ger97b, Ger99b]. During monitoring an experimenter can interactively select the set of events (and even the application objects) he/she is interested in. With the addition of the kernel-level events by JewelNT it can provide a view on the system that combines application semantics and information on its implementation by the system. A typical observation that can be made with JewelNT would be e.g. the overall time of an application's I/O operation and its breakdown into application specific activities (e.g. class-library calls), kernel activities (Win32 and system server threads), interrupt processing, waiting-time, and preemption by other threads.

The JewelNT monitoring system as depicted in Figure 2.1 consists of three components: the instrumentation, a remote communication infrastructure (both one instance per monitored node), and a central monitoring console with its graphical user interface.

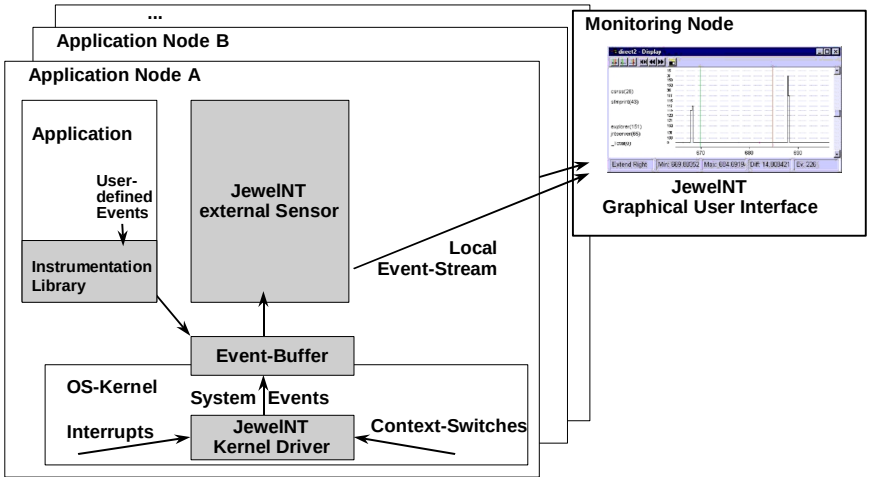


Figure 2.1: Components of the JewelNT distributed monitoring system for Windows NT.

JewelNT Instrumentation

The instrumentation augments the operating system to intercept all thread switches, interrupt handlers and other low level system-events. It inserts sensors that generate event-descriptions about these system events and stores them in a shared buffer (one on each observed node, typical size 128KB to 1 MB). By calling a library function from anywhere in the user-code, an application developer can also instrument his/her own software with additional user-defined events. They are placed in the same event-stream and exhibit arbitrary processing steps inside of an object implementation. Each event is stored with information about its type, a 64-bit high-resolution time-stamp, the executing CPU, process- and thread-id, and a 32-bit type-specific parameter.

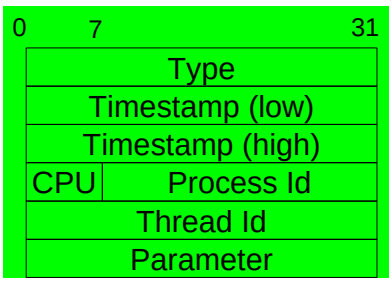


Figure 2.2: A JewelNT event-record

Figure 2.2 depicts such an event record. At this level an event-record does not contain a node-id. As all events in a local event-buffer would contain the same id this information is redundant here and can be added when the data is extracted from the buffer.

JewelNT addresses the above-mentioned implementation-related issues in the following ways:

1. **How to place a sensor to get aware of an event of interest?** Some operating system kernels provide the required hooks for adding instrumentation code at runtime. The VxWorks kernel e.g. provides special function-pointers as a global variable. If this variable contains a non-null value the kernel calls through this pointer a function on every event. A straightforward implementation will modify this pointer for inserting the sensor code. Similarly, the Windows NT debug kernel (so-called “checked kernel”) provides a kernel-level API that allows for the same approach. However, as the checked kernel contains all sorts of debugging code its timing behavior is significantly different from the normal kernel (so-called 'free kernel') and any real-life time-critical application will use the free kernel. This means relying on the checked kernel for instrumentation is not an option. Therefore, it becomes necessary to insert instrumentation into the existing binary of the free Windows NT kernel.

   This can be accomplished by driver-level programming. A Windows NT device driver coexists with the kernel in one address-space and on the same privileged execution level, i.e. driver code can modify the kernel image. The JewelNT kernel driver utilized this to patch the kernel and to insert the sensors. Basically, the sensor code is a detour though a C-function procedure provided by the driver. This function reads the time-stamp, collects the other event data, and writes the event-record to memory. As the debugging information available for the free Windows NT kernel provides sufficient information on the location of the central functions and variables of the kernel (e.g. for context-switching and interrupt-processing), this approach for instrumenting an operating system kernel is viable for all versions of Windows NT. However, it requires (once) careful hand coding of the generic instrumentation code in the driver, as it is critical to the reliability of the complete system.

2. **How to obtain time-stamps with sufficient accuracy?** Standard PC hardware, the primary target architecture of Windows NW, provides two timers that can be used for monitoring purposes: The build-in clock-chip that also generates the timer-interrupts and the on-chip performance counter, implemented by each Pentium-class CPU. While the first has the advantage of a standardized clock-rate (about 1 MHz), it has the drawback on modern processor with 1 GHz and beyond even this might be not enough and also that the provided timer is only 16 bits wide. Substantial software support is required

to emulate a 64-bit timer-register. The on-chip performance counter is clearly the first choice for time-stamping events. It runs at processor speed and it is 64 bits wide. It takes only one instruction to read its value and even on a SMP-machine the values of these registers are updated synchronously on all CPUs. Only on the most recent CPUs that use variable clock-rates for power management this counter doesn't provide a linear time-scale. But as this feature contradicts fundamentally to predictability of execution time, it can be neglected in the scope of real-time processing. An implementation problem that remains is the CPU-dependent rate of this timer. In a distributed system a central monitoring station needs additional information to determine the time-scale of each connected system under test. JewelNT provides this information by measuring the clock-speed once during installation on a new target system and sending this to the central monitoring station prior to any event trace data.

3. **How to minimize intrusion?** The JewelNT sensor code is designed obeying the following coding rules:
   - A local memory write is the only possible way of storing event data, direct file or network-access would take by far too much time.
   - Any subroutine-calls in the sensor should be replaced by inline code.
   - Any kind of system-calls (especially though call-gates) should be avoided.
   - Code-length, memory accesses, and locking-code should be minimized.

   Figure 2.3 shows JewelNT's buffer structure that stores the events and the sensor code. The buffer is organized as a ring-buffer, i.e. a FIFO where the sensors fill in new events and the external sensor extracts events and forwards them to further processing. As the buffer-structure is mapped into different address-spaces and to different locations, indices are used instead of absolute pointers for storing the current input- and output-positions. Note, that the function-lookalike statement `KeGetCurrentThread`(), `KeAcquireSpinLock()` and `KeReleaseSpinLock()`are actually short macros and that "spinlocks" are mapped to simple interrupt-level changes on single processor machines.

   On a 133 MHz Pentium machine this code executes in at most 2 µs. The execution time was determined by executing two subsequent sensors with flushed caches. Timing becomes much better if the code is already in the cache (typically 0.9 µs).

```
typedef struct {
    unsigned long spin_lock;
    unsigned long log_set;
    unsigned long max_length;
    unsigned long full;
    unsigned long input;
```

```
    unsigned long output;
    Event_Record rb[1];
}JewelNT_RingBuffer;

__inline Event_Record *jnt_sensor(
                  RingBuffer *rbp,
                  unsigned long e_type,
                  unsigned long e_param)
{
    void *cur;
    KIRQL old_irql;

    /* time stamp --> eax:edx */
    long t_high, t_low;

    if ((rbp->log_set & MZ_EV_TRIGGER) &&
        (e_type & rbp->log_set)) {
        KeAcquireSpinLock(&(rbp->spin_lock), &old_irql);

        if  (!rbp->full) {
            Event_Record *ev;
            ev = &(rbp->rb[(rbp->input)++]);
            rbp->input %= rbp->max_length;
            if (rbp->input == rbp->output)
                  rbp->full = 1;
            __asm { _emit      0x0F
                  _emit      0x31
                  mov        t_low, eax
                  mov        t_high, edx
            }
            ev->time_l = t_low;
            ev->time_h = t_high;

            KeReleaseSpinLock(&(rbp->spin_lock), old_irql);
            ev->type = e_type;
            cur = KeGetCurrentThread();
            ev->proc_id = *(long *)((long)cur + 0x1e0);
            ev->thread_id = *(long *)((long)cur + 0x1e4);
            ev->param = e_param;
            return ev;
        } else
            KeReleaseSpinLock(&(rbp->spin_lock), old_irql);
    }
    return 0;
}
```

Figure 2.3: The Jewel NT sensor code

5. **Where to collect event-traces?** The JewelNT kernel driver provides a memory buffer that is pinned to physical memory. This makes it accessible from anywhere in the kernel address-space. Note, that Windows NT has no further memory protection scheme inside the kernel. The virtual address of this buffer is propagated to all components inside the kernel by an entry in the global “system registry” database. The "ZwMapViewOfSection()" driver-API

is then used to map this buffer also into all address-spaces that contain sensor code. Kernel and user-level events are therefore written into the same buffer. This also simplifies the task of the external sensor and the central monitoring console, as only one buffer per machine has to be processed.

JewelNT Remote Communication Infrastructure

The remote communication infrastructure represented in Figure 2.1 by the JewelNT external sensor allows for the remote initialization and control of the measurement and the transfer of the local event-stream from each monitored node of the distributed system to the central monitoring console. It is implemented as a standard user-process that runs at low priority. It selects the instrumentation options and reads the local event-buffer located in shared-memory on behalf of the central monitoring console. If necessary the external sensor buffers events read from shared memory temporarily in the local file-system. The execution of this process and its file-system accesses are not on the critical path, i.e. if it is blocked due to an temporal overload the collected events are simply queue up in shared memory without any performance penalty for the generating thread. If the buffer runs full, additional events are discarded. In case of such a congestion of the monitor the observed activities are not effected, but a special event indicating the temporal buffer overflow is inserted in the event stream. This results in a warning on the graphical. The experimenter might react by increasing the buffer space.

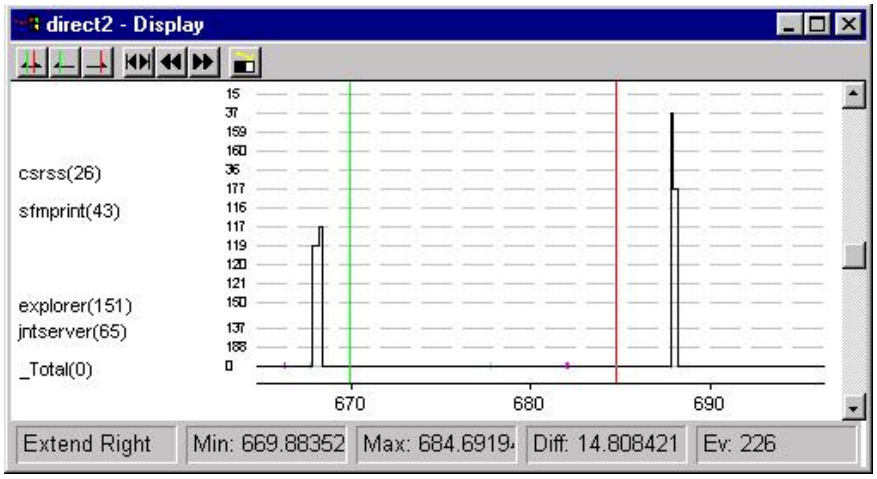


Figure 2.4: Gantt-chart display of JewelNT

JewelNT Central Monitoring Console

The remote communication infrastructure forwards event data from the monitored nodes to central monitoring console. On this node the experimenter can

manage the experiments remotely only using one graphical user interface. This interface allows selecting the interesting events, to start and stop measurements on a selected set of nodes, to initiate online data transfer to the monitoring console and to analyze the monitored data in detail. The analysis view displays either as text or as graphical display (shown in Figure 2.4) with variable zooming facility.

Process and thread assignment is displayed in a Gantt-chart. On the x-axis the time (in ms) is shown. The y-axis shows processes and their contained thread. For each CPU a line depicts the assigned thread at time *t*. All other events are shown in this display by marking the event-generating thread with a cross at the time where the event occurred.

Clock Synchonization

The JewelNT graphical interface allows controlling a distributed system under test from the central monitoring console. However, starting and stopping measurements on various nodes at about the same time doesn't already imply, that synchronized event traces are created. The JewelNT central monitoring console offers two mechanisms that allow for creating and visualizing synchronized traces:

For synchronization it uses the a-posteriori algorithm as described above. In order to apply the transformation of trace time-stamps "synchronous" events are required. Typically, receive events from broadcast networks can serve for this purpose. On most types of LANs (e.g. Ethernet) there is a known (and for coarser-grained measurements negligible) delay between the reception of the same frame at different nodes. In order to create the "synchronous" events, the central monitoring console sends broadcast messages containing a unique numbers. A sensor on each monitored node (implemented by the external sensor) detects these events and stores them in the local event-trace. If there are more than two common event pairs in two traces, the best synchronization can be achieved by choosing the pair with the maximum local time difference (as this minimizes the effect of inaccuracies of a single measurement). Once two pairs of such "synchronous" events are identified for two nodes a simple linear transformation of the local time of one of the involved nodes leads to global order between all events of these nodes (given that the local clocks have an unknown offset and a constant drift). Once two pairs of such "synchronous" events are identified for two nodes a simple linear transformation of the local time of one of the involved nodes leads to global order between all events of these nodes.

Transformation:

$$drift T_1 + offset = T_2$$

with

$$drift = \frac{t_1' - t_2'}{t_1 - t_2}$$

and

$$offset = t_1' - drift\, t_1$$

where t_1', t_1 and t_2', t_2 are the local timestamps of the two known synchronous events.

At the interface level JewelNT allows to link the Gantt-chart diagrams of two synchronized traces. With the linked cursors and scales it becomes easy to exactly determine the time difference between two events on different nodes.

Summary

A tool like JewelNT provides the infrastructure to observe an industrial operating system kernel in its application environment and to present the collected data online. In contrast to existing monitoring tools JewelNT allows to observe the activity of a number of distributed machines simultaneously and to correlate events between these interacting machines. This makes it suitable for the observation of distributed real-time systems at the operating system level. Beyond that, its ability to record and store not only the generic kernel events but also all types of events suggests its use also as the generic event collection infrastructure for user-level monitoring. However, as user-level code is highly application-dependent and cannot be traced by only a small set of generic sensors, tools and mechanism are required for simplifying the task of code instrumentation.

2.3 Instrumentation at Language Level

If observation wants to go into the details of the program execution, it is necessary to instrument the application itself. Adding and maintaining instrumentation by hand is a time-consuming and error-prone task. Thus, there is a need for an automatically instrumentation. Given that the major part of all software-projects is not yet described in a specification framework like UML that contains additional meta-information about the application, the main starting point for automatic instrumentation is the source code. Thus, the approach is *code* instrumentation. The source code contains the ultimate executable specification of the application and in real-time systems it is important to understand exactly the behavior of the executed code. A system that derives application instrumentation from a UML-like model is described in [Bec99].

This section discusses the possible options for automatic instrumentation and describes the tool „mc4p“ that implements automatic code instrumentation for a large class of object-oriented programs, namely those written in C++.

2.3.1 Issues in Code Instrumentation

The main principle of automatic code instrumentation is the utilization of the structural information of the program sources. Sensors are placed by a compiler using fixed syntactical rules. This results in very fine grained observation of nearly every step the software takes, but this also requires an event processing system that copes with the high bandwidth of event data. Even with a monitor that has been designed for low-interference like JewelNT, full code instrumentation still may lead to non-tolerable amount of interference with the system under test. Thus, a run- or compile-time filter is needed that reduces the amount of collected events, when they are not needed.

Procedural Languages

The main structural elements in a procedural programming language, like C or Pascal, are data types, variables, and procedures. Procedures themselves are structured by statements, blocks, and control structures. This enables the compiler to add sensor code before and/or after these units. With instrumenting only the procedure's entries and exits a number of features of the (not yet object-oriented) measurement model can be supported as listed below. Nearly all of the results of these measurements are hard to obtain by any other measurement technique:

- **Profiling**: An event-based system can obtain the same results as a traditional profiler. In addition the full population of all executions of a procedure is observed by event-based observation, leading to the distribution function, which may be important for evaluating whether an analytical model of the system is correct or not.
- **Statistics on a per thread basis**: Given that the thread identifier is added as a parameter to every event, all statistics can be displayed based on the relation (thread x procedure), rather than on a per procedure basis only. This is important as soon as not all threads that execute the same procedures are equivalent.
- **Tracing on a per thread basis**: The event streams provided by automatic instrumentation can be used for detailed tracing of the program's threads. The event traces can serve as input for visualization, for a step-by-step cost breakdown of complex operations. They are also valuable as a powerful debugging aid. Traces can help to explain bugs, like deadlock situations, or they can serve as a basis of a code coverage analysis.
- **A fully attributed call-tree**: As a superset of the three options mentioned above, the complete call-tree relation (procedure x procedure x thread) can be recorded. All members of the relation may have performance attributes. In the most complex case one attribute may contain the whole time-stamped calling-history, but often a counter or a distribution should be enough.

The *instrumented* Attribute

For complete observation it is desirable to add instrumentation to all procedures of a program, but often this may be not tractable because of the imposed interference, especially for very small and short procedures. To allow for a more selective instrumentation, each procedure is tagged by an additional attribute *instrumented* that defines during compilation time whether the procedure should contain the additional sensor code or not. The actual value of the *instrumented* attribute may be defined either within the source (by a new keyword or a comment that proceeds the procedure declaration) or by an external definition list that contains the names of all *instrumented* procedures. The first method has the advantage that all information concerning the sources is kept consistent within one file, whereas the second possibility allows to leave the original source file completely unchanged. Both, the additional keyword and the definition file are easy to skip, when no instrumentation is needed, and they both do not reduce the readability of the sources (like e.g. #ifdef's would).

The *observed* Attribute

Another option to reduce the overhead of instrumented code, when no observation is required, is an additional run time filter mechanism that determines whether events created by a certain procedure are observed or not. This filter should be employed in a very early stage of event processing (best within the generating code) to avoid any waste of bandwidth and keep perturbation as small as possible. This means in addition to the *instrumented* attribute that is evaluated during compile time each *instrumented* procedure obtains an additional boolean attribute *observed* that is evaluated during run time. If a procedure is *observed* its events will be reported, otherwise they are discarded. The *observed* attribute may be changed during run time either by the observing event processing system or by the system under test itself. The first option requires that the event processing system has access to these attributes (as implemented with *relevant-table* in the JEWEL system [Lan92]), while the second implies that the system under test is written to be aware of its own instrumentation. The *observed* attribute of procedures can be stored in a static data-structure as the number and names of the procedures are known at compile (or link) time.

Object-Oriented Languages

Object-oriented languages add a number of features to the procedural programming paradigm. Three concepts, namely the notion of classes, instances, and attributes are of high value for automatic code instrumentation. Other common features of object-oriented languages, like function overloading and polymorphism do not affect instrumentation substantially.

Classes introduce a new important structural element that provides a lot of additional information for automatic instrumentation. As a program written in an object-oriented language should be a one-to-one mapping from an object-

oriented design, the measurement model supported by an automatic instrumentation of object-oriented code is more likely to match the original system model. Classes also represent a higher-level abstraction for event filtering than simple procedures do. An experimenter may now specify the name of a class he is interested in, instead of a bunch of procedures or a source file with a number of related procedures.

A new quality for code observation is introduced by the concept of instances of classes. As every method of a class knows implicitly about the object it manipulates, this knowledge can be made visible to the observing system, by generating instance-related events. In a procedural language instances have to be specified explicitly within the procedure parameters or, even worse, in global variables. In both cases it is generally impossible for automatic instrumentation to name the instance on which the procedure operates. The improvement when observing object-oriented programs is, that events (and thus also performance indices) can be collected, filtered, and displayed on a per instance level rather than on a per procedure level. This enables a number of additional features of the generic measurement model:

- **Statistics on a per instance basis**: In many cases it is misleading to display information on a per function or per class basis, as objects of the same class may have completely different characteristics due to their actual instance (e.g. statistics about context switching of the class "process" does not say much about the scheduling behavior of a certain process).
- **Observation of instances in a certain context**: One might be interested in the behavior of instances in a certain context or subsystem, rather than in all instances of the same class (e.g. buffers within a certain protocol, but not within the rest of the system).
- **Probe observation**: In order to get a better understanding one might be interested in only generating events for some probe instances rather than the whole population (e.g. measuring the delays for a sample request on its way through the communication system).
- **Attribute Traces**: The value of some attributes of some objects at certain points during the execution may be traced. E.g. the afterimages of some important status variables may be reported when a method of this object has been executed.

Instrumentation and Classes

A class collects all methods (or member functions) and status variables (attributes) that define a certain object-type. Instrumenting classes means inserting sensors into the methods' code in order to signal object invocations, their parameters and attribute value changes. In a procedural language only procedures have an instrumented attribute. In an object-oriented language the instrumented attribute can be extended to classes, in order to collectively enable or disable the

observation of methods of a certain class. The following definition of the relation between the instrumented attribute of classes and methods allows to instrument a single method as well as complete classes.

If a class *x* is *instrumented*, all contained methods or classes of *x* are implicitly *instrumented*. If only a contained class or method is *instrumented* the surrounding class is not necessarily *instrumented*.

In order to be flexible automatic instrumentation should only determine the places where to put the sensors, but not the semantics of the sensors itself. The feature of generating events can be regarded as a certain property of all *instrumented* classes. Thus it is natural within an object-oriented environment to express this property as a separate base class. This abstract base class, called *_instrumented_class*, defines the protocol for the inserted sensors. Their actual implementation is encapsulated in a descendant of the abstract base. Each class that contains at least one *instrumented* method has to inherit from a suitable descendant of *_instrumented_class*. Now the process of automatic instrumentation only has to insert the calls to sensor methods known from the base *_instrumented_class*.

Instrumentation and Inheritance

In an object-oriented language the behavior of a class is not only described by the methods that are defined within the class itself, but also by inherited methods from other classes. Thus, the *instrumented* attribute has to be extended to the inheritance relation as well. The following extension to the definition above describes one possibility that enables a high degree of flexibility and requires only a small amount of additional information besides the source file.

If a class *x* inherits from another class *y* all methods and subclasses that are *instrumented* in *y* are *instrumented* in *x* as well.

Instrumentation and Instances

To allow for the detailed observation of single instances without causing to much interference during observation, it is of a high value to extend the *observed* attribute from procedures to instances (or even methods of instances). One restrictions for this is obvious:

An instance can only be *observed* if at least one method of the class it belongs to is *instrumented*.

Otherwise there is nothing to report from an *instrumented* instance. But as instances are created at runtime, it is generally not possible to distinguish statically between *observed* and not *observed* instances. This problem can be solved by adding the *observed* status of an instance to the according implementation of *_instrumented_class*, either simply one boolean per instance or, even more detailed, one a per *instumented* method. Thus the storage allocation and the initiali-

zation of the per instance attributes are done dynamically during the creation of new objects. As the implementation of the *observed* attribute is now hidden in the implementation of *_instrumented_class*, an automatic instrumentation tool does not have to deal with this attribute; it only has to supply enough information, i.e. about the instance and the method to each sensor. Again, this is known statically during compilation and it can be hidden in the event-identifier. The constructor of _instrumented_class is provided with type information about the class it belongs to in order to allocate the correct data structures for keeping track of the observed status of an object.

Instrumentation of Attributes

To provide information about the internal status of an object, its attributes have to be made visible. In an event-based system this can be done by creating an event that contains the value of the attribute in its data part. But when should such an event be created? Whenever the value of the attribute has been changed, continuously with a certain sampling rate, on request of the experimenter, or during certain steps of the execution? The first option seems to be the most effective, since it provides a complete trace with a minimum amount of bandwidth. But as attributes may be changed by arbitrary references this approach requires either help from the memory-management to detect write access to certain locations or a major code change to check every reference before any data is written. Continuous sampling of a larger number of attributes requires a lot of additional cycles and there is still the chance of missing rapid changes, while the "on request" option does not even allow to reconstruct a global snapshot of the systems state. Thus the last option is promising, as it can be combined with the already discussed sensors. One approach is to add the value of the *instrumented* attributes of an object to every method end event that is produced by this object. This will result in a complete trace of all afterimages of all *observed* method calls without producing additional events.

2.3.2 The mc4p Tool

In order to prove the usefulness of automatic instrumentation a tool has been built that implements this approach [Ger94]. This tool has to fulfill several requirements. It should be easy to use, applicable to a large class of existing programs, easy to adapt to different event processing systems, and independent of special compilers. These requirements lead to a design of a preprocessor, called *mc4p* (Martins C Plusplus Preprocessor) that translates a source written in K&R C, ANSI C, or C++ into the same language again, but adds configurable instrumentation statements. This allows for an easy integration into the normal building process of any C/C++ source with an arbitrary compiler. As mc4p only defines the places in the source, where the additional instrumentation statements will go, but not the actual statements, it is open to cooperate with processing systems other than JEWEL.

The mc4p tools implements an automatic instrumentation according to the measurement models as described above. This means mc4p keeps track of the *instrumented* attribute of classes and methods, and attributes. It implements the name space management as described above and it applies three kinds of modifications to the source files: it adds the generic instrumentation code, hidden in a descendant of *_instrumented_class*, it inserts the sensors that indicate start and stop of methods, and it adds attribute reporting sensors to the end of *instrumented* functions. While mc4p is able to instrument at C/C++-block level as well, no attempts have been made so far to optimize sensors placement as in [Mah01].

Instrumentation of Procedures

The automatic instrumentation of a program written in a procedural language is straightforward. The complete structure of the call-tree can be described by events, if every entry of any procedure produces a unique event. Additionally the timing of a procedure can be observed if another event also indicates a return from a call. This can be obtained either

1. by adding code within the called procedures body,

```
proc_a(){
  _method_start(proc_a);
  .
  .
  .
  _method_end(proc_a);
}

proc_b(){
  proc_a();
}
```

2. by instrumenting each call sequence,

```
proc_a(){
  .
  .
  .
}

proc_b(){
  _method_start(proc_a);
    proc_a();
  _method_end(proc_a);
}
```

3. or by wrapping each call in an instrumented stub.

```
proc_a'(){  // was proc_a ()
  .
  .
  .
}

proc_a(){
  _method_start(proc_a);
    proc_a'();
  _method_end(proc_a);
}

proc_b(){
  proc_a();
}
```

With the first method instrumentation can be done statically as long as the source text of the procedure is available. The second possibility also works for external procedures (e.g. in a library) but it requires a dynamic determination of the procedures name as a call may be done via a reference. The third option is a kind of a mixture of the other two possibilities, as it can be done statically and it also works for external procedures. The major drawback of this method is the slightly increased overhead if the call to the wrapper really introduces a second call sequence. Implementing the wrapper as an open procedure will avoid a lot of the additional code.

Instrumentation at the beginning and the end of a procedure body as well as wrapping of procedures can be implemented at preprocessor level. As there is only one starting point of each procedure, it is easy to insert a sensor in front of the first statement of a procedure body. The end of a procedure has to be reported either before a return statement is executed or when the textual end of a procedure is reached. A problem arises when a complex computation is done within the expression of the return value. If the end of the procedure is signaled before the return expression is evaluated, the time consumed by longer computation will be accounted for the calling procedure and if other procedures are called within this computation even the call tree will be mixed up. This problem can be solved by inventing an additional temporary variable *x* of the procedures return type and changing a sequence "*return expr*" into "*x := expr; _method_end(); return x*". Simple C procedures are instrumented by mc4p using body instrumentation technique.

In C++ another, more elegant way, of body instrumentation can be implemented. In C++ local variables (allocated on the stack) are automatically constructed (the constructor is called) *before* a procedure starts and destroyed (the destructor is called) *after* the procedure has returned. This mechanism is implemented by the

compiler. To use this mechanism for signaling start an stop events a new class *sensor* has to be defined. In its constructor this class signals the start of a procedure and in its destructor it signals its end. If now a variable of this class *sensor* is added to an instrumented procedure, the compiler will arrange exactly the code we need, as it signals the start before the first instruction of the procedure is executed and it signals the end even after the return expression has been evaluated. The latest version of mc4p uses this kind of body instrumentation for C++ instead of the (more C-like) version described above, but the two version functionally equivalent. (The example described below uses the C++ method)

Wrapping requires a redefinition of the procedure that has to be instrumented. The body of the wrapper simply contains the instrumentation indicating the start and the end of the procedure and a call to the original code with unchanged parameters. A name clash between the original procedure and the instrumented stub can be avoided by renaming the original procedure and naming the stub like the unchanged original procedure. The same technique as above, using a temporary variable, can be used for propagating the return value of a wrapped procedure. The mc4p tool has to use the wrapping for instrumenting inherited C++ methods.

The value of formal parameters of procedures can be determined and signaled easily during the start sequence of a procedure. Before the start of a procedure a sensor is inserted the mc4p preprocessor adds per *instrumented* parameter a sensor that reports its current value. If the implementation allows for collecting events and processing them in a batch-like manner, the sensors may condense the parameter values and the procedure start into a single event.

The parameter sensors have to know about the event_id, a pointer to the parameter, the length of its binary representation, and a type identifier. While the pointer and the length are useful for fast internal copies, the type identifier will allow for encoding the data into a hardware independent representation (e.g. using XDR). This is generally important for interpreting the values outside the context of the C++ program. Type information is also necessary when transferring the values in a heterogeneous distributed environment. The type identifiers may cover either only the basic types, or in a more advanced implementation also constructed types like complete classes.

Instrumentation of Classes

Whenever mc4p has to insert a sensor it inserts one of the methods of *_instrumented_class*. The declaration of *_instrumented_class* is shown in Figure 2.5. A descendant of *_instrumented_class* has to be introduced as a new base class of every class that contains at least one *instrumented* method. It has to be insured that every instance contains the components of this base class only once, even if the class inherits from one or more other classes that are already derived from *_instrumented_class*. C++ provides the virtual inheritance concept to achieve exactly this.

All static information about the location of the sensors can packed into the event identifier as described above. The dynamic parameters, like thread identifier or object identifier are determined, whenever needed, within the implementation of *_instrumented_class*. This is possible because the object identifier provided with the C++ *this* pointer is always known within the implementation of *_instrumented_class*. All status that has to be stored or cached within a sensor can be implemented as member variables of a descendant of *_instrumented_class*. E.g. a simple collection mechanism for reducing bandwidth might combine a *start_method* and an *end_method* event to one event that signals the duration of a certain method invocation. A timestamp of the *start_method* event can be stored within the object until the according *end_method* event occurs.

```
class _instrumented_class {
public:
     _instrumented_class() {};
     ~_instrumented_class() {};

// method start and end sensors
     virtual void
     _start_method(unsigned long event_id) const = 0;
     virtual void
     _end_method(unsigned long event_id) const = 0;
     virtual void
     _end_constructor(unsigned long event_id) const = 0;
     virtual void
     _start_destructor(unsigned long event_id) const = 0;

// parameter sensor
     virtual void
     _add_method_local(unsigned long event_id, void * ptr,
                       int size, typeinfo type) const = 0;

// attribute sensor
     virtual void
     _add_attr(unsigned long event_id, void * ptr,
                       int size, typeinfo type) const = 0;

// collector for attribute sensors (reimplemented by every
//  instrumented class)
     void
     _report_attr() const {};
};
```

Figure 2.5: Declaration of instrumented_class

Instrumentation of Methods

For instrumenting the beginning and the end of methods the mc4p tool uses the techniques for procedures as described above. It either instruments the body of a method when it is newly defined within an *instrumented* class or it uses wrap-

ping to allow for the instrumentation of inherited methods that where not *instrumented* in their defining class. This leaves the original class and its methods unchanged (not *instrumented*) and allows to use the new wrapper with the same functional behavior in a derived class in an *instrumented* version (see section 6 for an example). An exception to the standard rule for inserting sensors in methods has to be considered for class constructors and destructors, as the code of their body does not denote the actual start and end of their execution. All constructors and destructors of inherited classes are called before, res. after, the execution of the constructors or destructors of the current class. To get a realistic timing, instrumentation rules for these special methods are slightly different. Only the end of a constructor (res. the beginning of a destructor) is instrumented in the way described above. The start of a constructor and the end of a destructor is signaled by the according methods of *_instrumented_class* itself, as they are executed as first or last part of construction or destruction of the composed object (given a proper ordering of the inheritance list – if there are other virtual base classes than *_instrumented_class* and if they are located deeper in the inheritance hierarchy, these are initialized before and destroyed later)

Instrumentation of Attributes

The mc4p tool implements instrumentation of attributes by creating a new method, called *_report_attr()*, for every class that contains at least one *instrumented* attribute. The *_report_attr()* function generates events that report the values of all instrumented attribute of this object. In order to do this the *_report_attr()* function in turn calls the *_add_attr()* method for each attribute. The *_add_attr()* function is defined by *_instrumented_class* or one of its descendants. It produces an event, containing the current value of an attribute and it works similar to the parameter sensor described above. As the afterimage of an object-invocation should be reported, the *_report_attr()* function is called right before the call to the *_end_method()* method. Again, the various attribute values and the method termination event may be condensed into a single event.

A similar mechanism as for observed attributes is used by mc4p to allow for instrumentation of global variables within a C program. A complete instance of a program can be regarded as a single object and the global variables as its attributes. Corresponding to *_report_attr()* and *_add_attr()* within C++ objects mc4p uses the free procedures *_report_global()* and *_add_global()* to report the current status of all *instrumented* global variables. These procedures are defined locally once per source file and they are called at the end of all procedures.

Using mc4p

For instrumenting a program, mc4p has to be inserted into the building process as shown in Figure 2.6. After a source file has been successfully compiled using the usual C/C++ compiler, mc4p has to be run on the output of the normal C

preprocessor. Mc4p's output, the instrumented program, then has to be compiled using the normal C/C++ Compiler.

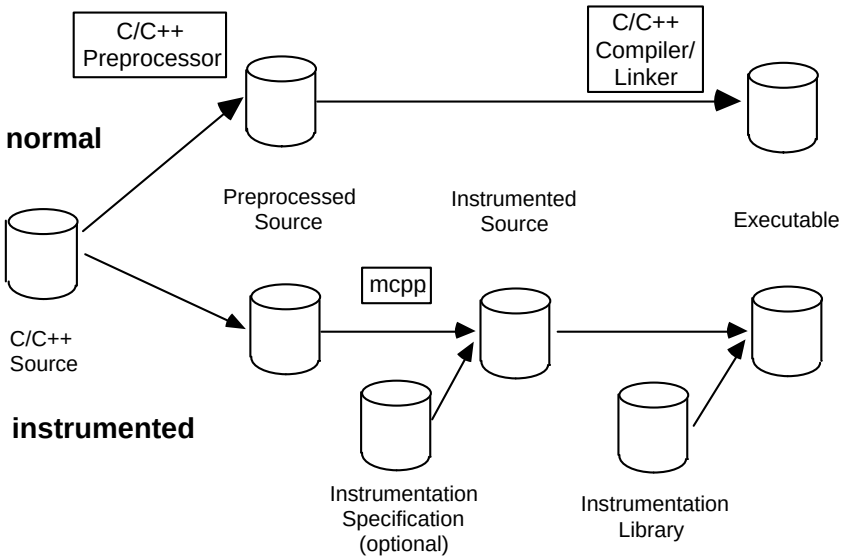


Figure 2.6: The Building Process using mc4p

Event Name Management

Event names are generated in a global name space containing the system under test-, class-, and method- and attribute-identifier plus the event type as described above. These names are mapped to integers that are actually used during data transmission. The mc4p tool implements a name server for events as a separate server process and communicates with the server via RPCs. This allows for concurrent distributed compiler sessions that modify the same sub-tree of the name space, as it is common for a group of cooperating software developers. To allow for a persistent name space that survives independently of the server, the current state of the name space can be flushed to a file.

An attempt to (pre-)compile a modified version of the same example code again will result in the same events identifiers for all events that are still in place. New events are added to the database automatically, but old events are never discarded without user intervention. As event names are never deleted or changed dynamically every client of the name server is allowed to cache event names and mappings as needed without any invalidation mechanism. This will speed up event processing components significantly.

Of course, the name server does not only resolve class and method names into event identifiers as requested by the mc4p during compilation. It also maps in the

other direction event identifier to classes and methods as required by all tools do event processing based on the object-oriented structure of the observed system.

Example

The mc4p tool has been implemented and tested with a large number of C++ programs. The preprocessor runs on Unix or Windows NT workstations. The largest test so far has been carried out by translating an application that includes the complete Microsoft Foundation Classes library (more that 50 000 lines of code) and instruments the derived application main window class.

In the following a basic example of a mc4p-generated instrumentation is given. The code in Figure 2.7 shows a C++ implementation of the well known *stack* class in a rather simple implementation plus a derived class *xstack* that implements a stack with an *is_empty()* method. The classes *stack* and *xstack*, the parameter *i* of the *stack::push()* method, the attribute sp (the stack pointer of *stack*) and the free procedure *main()* have been marked to be *instrumented*. In order to keep the example simple the instrumentation has been defined within the original code by using the additional keyword **instrumented**. As long as no instrumentation is needed, **instrumented** is simply removed by the ordinary C++ preprocessor.

Figure 2.8 shows the output generated by mc4p. There are a number of modifications, marked in boldface:

- The inheritance list of all classes that include any instrumented code are modified to contain *_inst_log* as the first base class. In the case of the *stack* class the inheritance list is newly created while the list of xstack has been simple extended. The *_inst_log* class has been derived from the abstract class *_instrumented_class*. It implements the log style print-out of the produced events as shown in Figure 2.9.
- All methods that are instrumented and not inherited from other classes now have an additional local variable **_s** of type **_sensor** or **_sensor_constr** with one or two integer parameters. Within the constructor and the destructor of these variables the methods of *_instrumented_class _start_method()* and *_end_method()* are called. The parameters of these _s varuables are the event_ids that are forwarded to the *_start_method()* and *_end_method()* calls.
- All free procedures (in this case *main()*) are surrounded by calls to the procedures *_start_procedure()* and *_end_procedure()*. These procedures are not methods of *_instrumented_class* but free procedures themselves. Thus, they do not take this-pointers and can be used for instrumenting free procedures and static methods.
- The additional method *_report_attr()* has been redefined in all *instrumented* classes to report the values of the *instrumented* attributes using _add_attr().

In addition calls to *_report_attr()* are added to the exit sequence's of all *instrumented* member_functions. The type identifier as a parameter of *_add_attr()* as described above is not yet implemented in the current version of mc4p, thus only binary copies of attributes are supported.

- Finally, for every *instrumented* formal parameter mc4p has inserted a call to the *_add_method_local()*. It is executed in the start sequence of the functions right before the call to *_start_method()* is done.
- All inserted sensors receive an event identifier as parameter. The mapping between these identifiers and the semantics of the event is maintained in the mc4p event name database.

```
#include "inst_log.h"

const int MAXSTACKSIZE = 100;

instrumented class stack {
private:
instrumented       int sp;
           int array[MAXSTACKSIZE];
public:

     inline stack() {sp = 0;};
     inline ~stack() {};
     inline void push (instrumented int i) {array[sp++] = i;};
     int & pop (void);
     virtual int pop (int &);
};

int &stack::pop(void) {
     sp--;
     return array[sp+1];}

int stack::pop(int &i) {
     if (sp == 0)
            i = -1;
     else
            i = array[--sp];
     return i;}

instrumented class xstack: public stack {
public:
     int is_empty(void);
};

int xstack::is_empty(void) {
int h;
     if (pop(h) != -1) push(h);
     return (h == -1);
}

instrumented void main() {
     xstack S;
```

```
        for (int pop=1; pop < 2; pop++)
                S.push(pop);

        while (! S.is_empty())
                S.pop();
    }
```

Figure 2.7: The stack example prepared for instrumentation with mc4p

```
    // some header code
         :
         :

    const int MAXSTACKSIZE = 100;

    class stack: virtual public _inst_log  {
    private:
         int sp;
         int array[MAXSTACKSIZE];
    public:

         inline stack(): _inst_log(0x1000, "stack") {_sensor_constr
_s((_instrumented_class*)this, 0x1002); {sp = 0;}};
         inline ~stack(){_sensor_destr _s((_instrumented_class*)this,
0x1003); {}};
         inline void push ( int i){_add_method_local(0x1004, &(i),
sizeof(i)); _sensor _s((_instrumented_class*)this, 0x1005, 0x1006);
{array[sp++] = i;}};
         int &pop (void);
         virtual int pop (int &);
    inline virtual void _report_attr() const
    {
         _add_attr(0x1007, (void *) &(sp), sizeof(sp));
    };
    };

    int &stack::pop(void){_sensor _s((_instrumented_class*)this,
0x1008, 0x1009); {
         sp--;{
         return array[sp+1];}}}

    int stack::pop(int &i){_sensor _s((_instrumented_class*)this,
0x100a, 0x100b); {
         if (sp == 0)
                 i = -1;
         else
                 i = array[--sp];{
         return i;}}}

    class xstack: virtual public _inst_log,  public stack {
    public:
         int is_empty(void);

    inline virtual void _report_attr() const
    {
    stack::_report_attr();
    };
```

```
   public:
   xstack():_inst_log(0x100c) {_sensor_constr
_s((_instrumented_class*)this, 0x100d);};
   };

   int xstack::is_empty(void){_sensor
_s((_instrumented_class*)this, 0x100e, 0x100f); {
   int h;
        if (pop(h) != -1) push(h);{
        return (h == -1);}
   }}

    void main(){_start_procedure(0x1010); {
        xstack S;{

        for (int pop=1; pop < 2; pop++)
                S.push(pop);

        while (! S.is_empty())
                S.pop();

   }_report_global(); _end_procedure(0x1011);}

   static void _report_global()
   {
   };
```

Figure 2.8: The instrumented stack example

Figure 2.9 shows a trace of a test run of the example program using a descendant of *_instrumented_class* that simply writes ASCII text into a log-file. In this case the timing of the program is completely dominated by the costs for the text output. Sensor versions that directly use JewelNT user-level API for writing to the memory-mapped have been implemented and provide the desired low-interference property [Bec99].

To get a lower bound for the interference of the monitored program, the costs of a call to an empty sensor of _instrumented_class have been analyzed. On an Intel Pentium the Microsoft C++ Compiler generates 7 additional instructions per sensor. This includes the computation of the object's *this*-pointer and the virtual call mechanism of the sensor's method.

```
 void main ( ) () starts
    xstack(0): xstack::xstack ( ) () starts
    xstack(0): [int sp = 0x0(size 4)]
    xstack(0): stack::stack ( ) () ends
    xstack(0): [int sp = 0x0(size 4)]
    xstack(0): xstack::xstack ( ) () ends
    xstack(0): stack::void push ( int ) (int i = 0x1(size 4)) starts
    xstack(0): [int sp = 0x1(size 4)]
    xstack(0): stack::void push ( int ) () ends
    xstack(0): xstack::int is_empty ( ) () starts
       xstack(0): stack::int pop ( int & ) () starts
       xstack(0): [int sp = 0x0(size 4)]
       xstack(0): stack::int pop ( int & ) () ends
```

```
       xstack(0): stack::void push ( int ) (int i = 0x1(size 4)) starts
       xstack(0): [int sp = 0x1(size 4)]
       xstack(0): stack::void push ( int ) () ends
    xstack(0): [int sp = 0x1(size 4)]
    xstack(0): xstack::int is_empty ( ) () ends
    xstack(0): stack::int & pop ( ) () starts
    xstack(0): [int sp = 0x0(size 4)]
    xstack(0): stack::int & pop ( ) () ends
    xstack(0): xstack::int is_empty ( ) ()
       xstack(0): stack::int pop ( int & ) ()
       xstack(0): [int sp = 0x0(size 4)]
       xstack(0): stack::int pop ( int & ) () ends
    xstack(0): [int sp = 0x0(size 4)]
    xstack(0): xstack::int is_empty ( ) () ends
    xstack(0): stack::~ stack ( ) () starts
    xstack(0): destroyed
 void main ( ) () ends
```

Figure 2.9: A trace with the inst_log class of the instrumented stack example

Figure 2.10 finally shows a small part of the dump of mc4p name-server after processing the sample program. The entries are organized in a tree with five levels:

1. The system under test (in this case “Defaul_SUT”).
2. The class (here “stack” and “xstack”).
3. The component-types (variable, base-class, constructor, method, event).
4. Instances of the components (the actual variables, base-classes, constructors, methods), their instrumentation status (“i” = instrumented, “u” = not instrumented), and their first occurrence in the code (file-name and line number). In case of events this level describes the names of the entities that the events belong to (e.g. the method-names)).
5. The event numbers and their semantics (start, end, value).

```
0     Default_SUT         1012
. . .
1              stack
2                         Variable
3                                   int'sp'   i:"test.cpp":7-7
3                                   int'array'['MAXSTACKSIZE']' u:"test.cpp":8-8
2                         Constructor
3                                   stack'('')'          u:"test.cpp":11-11
2                         Method
3                                   ~'stack'('')'        u:"test.cpp":12-12
3                                   void'push'('int')'   i:"test.cpp":13-13
3                                   int'&'pop'('')'      u:"test.cpp":18-20
3                                   int'pop'('int'&')'   u:"test.cpp":22-27
2                         Event
3                                   stack'('')'
4                                             start_constructor   1000
4                                             end_constructor     1002
3                                   ~'stack'('')'
4                                             start_destructor    1003
3                                   void'push'('int')'
4                                             int'i'    1004
4                                             start     1005
```

```
    4                                       end       1006
    3                               int'sp'
    4                                       value     1007
    3                               int'&'pop'('
    4                                       start     1008
    4                                       end       1009
    3                               int'pop'('int'&')'
    4                                       start     100a
    4                                       end       100b
    1           xstack
    2                   Method
    3                               int'is_empty'(')'   u:"test.cpp":34-38
    2                   Baseclass
    3                               stack     u:"test.cpp":32
    2                   Event
    3                               xstack'(')'         0
    4                                       start_constructor   100c
    4                                       end_constructor     100d
    3                               int'is_empty'(')'   0
    4                                       start     100e
    4                                       end       100f
```

Figure 2.10: A dump of the name space as generated by mc4p

This information is kept persistent by the mc4p name server and it assures that event names a constant over the lifetime of a software project. Also it illustrates the amount of structural information that is extracted by mc4p.

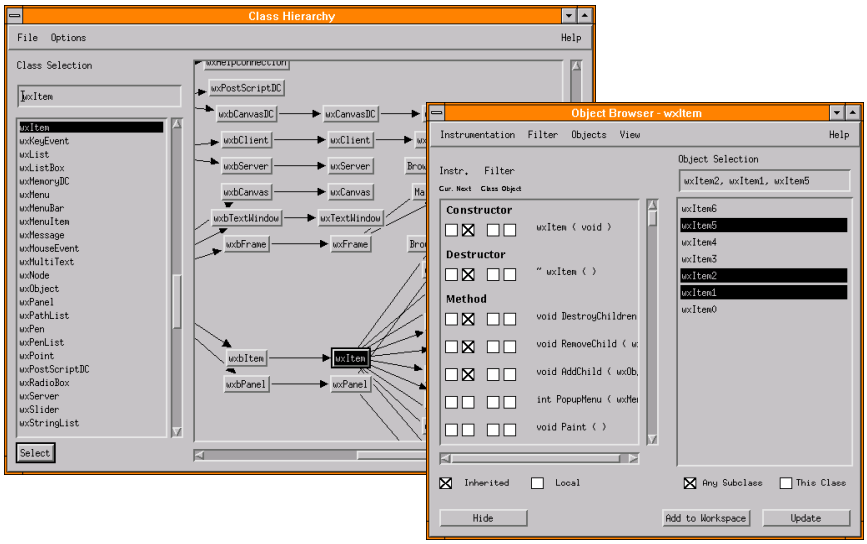


Figure 2.11: Sceenshots of a class-browser and a visual instrumentation tool based on the mc4p name server

Based on this information also graphical tools for class-browsing and visually guided code instrumentation have been developed as educational projects (sceenshots are show in Figure 2.11).

2.4 Instrumentation at Middleware Level

Before going into the details of instrumentation and monitoring of the middleware level, it first has to be clarified what middleware actually is. In the IT-world there is a quite fuzzy understanding of this term and this results from the fact that "middleware" actually denotes three different categories of software [Rac01]:

1. *Presentation middleware* only cares for displaying data remotely. A Web browser and server communicating via the HTTP protocol (hypertext transfer protocol) can for example be classified into this category.

2. *Database middleware* is deployed to access database management systems remotely. For example, SQL requests being sent to the DBMS and transferring back the results to the client are a typical task for database middleware.

3. *Application middleware* is used to distribute the application logic, and therefore functions as a general purpose programming platform for distributed applications. Its goal is to enable application programmers to build interacting components using middleware to abstract from given system details.

For the remainder of this thesis the term middleware will be used the sense of application middleware. Among the various solutions for application middleware, including PVM (message-based) or ONC or DCE (client/server-based), the focus is solely on object-oriented approaches like CORBA or DCOM. Therefore, the definition for middleware in this thesis is:

Middleware: A software layer between operating platform and application that enables the interaction of potentially distributed objects, aiming at transparency and independence from the surrounding runtime environments.

Since the rapid success of object-oriented middleware like CORBA [OMG95] or DCOM [Edd99] objects do not just reside locally inside of user processes but they are now visible entities in a distributed system. These middleware-objects are usually bigger than standard C++ objects, as they act as clients and servers in (possibly) cross-context or cross-machine invocations. Bigger means not necessarily bigger in terms of code size or internal status (while a complete database can be hidden behind one interface object), but the execution time of middleware-objects is usually big enough to justify a full-fledged remote invocation via the network. Even with today's high-speed networks this is still in the order of some microseconds and thus several orders of magnitude larger than an intra-context C++ invocation. Also for these objects client/server-style computing requires explicit assignment of network (RPC-style messages), memory (buffer-space), and CPU resources (server-threads). This means middleware-objects as

units of distribution and scheduling are ideal entities for adjusting and adapting real-time behavior and thus they are also of special interest for monitoring in a time-aware system.

The previous section has described how language-level objects can be instrumented using compiler techniques. Similar techniques could be applied for instrumenting middleware-objects as well. The interfaces of middleware-objects are usually defined in an *Interface Definition Language* (IDL) that is compiled into *stub*-code that is later liked against the implementation. A modified IDL compiler could insert sensor statements into the generated stub-code, just as mc4p does for C++ code. However, a major drawback of this approach is, that it can only be applied to source-code. One of the ideas of object-oriented middleware however is the component model that allows to integrate third-party object-code and problem-specific source code into one distributed application. Especially this integration of black-box third-party code introduces one of the biggest problems in object-oriented real-time computing, as also the timing of these components is not open for further analysis. Time-awareness through monitoring is an approach to tackle this problem. This and the fact that invocations of middleware-objects are routed through a common runtime-system demand and allow for a different kind of instrumentation. Like at the operating system level, a generic instrumentation of just a few important events of interest in the runtime-system can provide the required information. The next section tries to identify these generic events of interest.

2.4.1 The Activity Concept

The basic abstractions provided by a distributed object-oriented middleware framework are the same as in an object-oriented language, namely classes, objects and (location transparent) method invocations. However, execution model in a distributed object-oriented environment differs from that of a local C++ program. In the local case the computation is driven by a number of threads. A thread starts in the context of a certain object and if this object invokes another object the thread switches into the context of this object. These invocations can be nested, but at any nesting level the thread (and its Id) identifies the complete chain of invocations that belongs to one top-level computation.

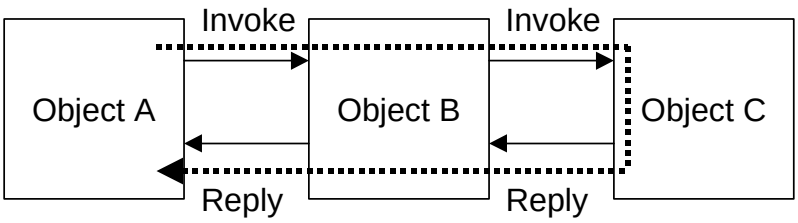


Figure 2.12: An activity originating from object A with nested invocations of objects B and C

Things are different in a distributed environment. Here each object can have a different mapping of invocations to serving threads and this mapping changes each time another object is called. Without additional measures the pattern of object invocations in such an environment is flat and reveals no nesting hierarchy. From the pure functional point of view this causes no problems. However, in real-time-systems it is usually only the top-level computation that is associated with parameters like deadlines, periods, or priorities. Thus, for a correct execution it becomes important to know which top-level computation is responsible for a certain (nested) invocation. Transferring the thread model to a distributed system yields the concept of an activity: an activity in a distributed object-oriented system is as a distributed sequence of possibly nested method invocations (see Figure 2.12). An activity branches from a top-level object and returns control to the same top-level object. It can contain nested method calls, each of which has well defined start and end points. By invoking other objects, an activity can cross object boundaries and site boundaries in distributed systems. Considering real-time requirements, the end-to-end timing behavior of an activity is clearly given by the execution times of the activities top-level method invocations. Thus, a real-time activity is an activity whose top-level method invocations are subject to real-time requirements.

The activity concept (without considering monitoring and real-time aspects) already has been implemented in some distributed object-oriented systems and distributed transaction systems, e.g. [Cah93]. However, it is not part of the specification of neither CORBA nor DCOM. The first standard that contained it was the Real-Time CORBA specification [OMG99b, Sch00]. It states that an abstract activity is represented in an ORB by concrete entities: a message within a transport protocol, a request held in memory, and a thread scheduled to run on a processor. These three phases are termed “in-transit”, “static” and “active” respectively. Real-Time CORBA provides the ability to effect these three phases of an activity. It leaves the developer to delimit their concept of an activity by the way they coordinate these concrete entities using the interfaces specified. The Real-Time CORBA Scheduling Service provides abstractions to work in terms of activities.

2.4.2 Monitoring Activities

As activities are the units of timing specification and already supported by middleware-frameworks, they are also the natural units for being monitored. What is needed to capture the events of activities in a distributed object-oriented environment?

“Active” activities are executed by local threads. This means whenever the thread that carries an activity crosses an object boundary a sensor has to generate an event. In contrast to pure local invocations now it becomes important to keep track which invocation and which thread acts on behalf of which activity. Only if this mapping is recorded in the event trace, the complete invocation path of an activity can be followed by the monitor. This can be accomplished by storing not just the thread Ids in the event records but also "activity" Ids. However, as activities and their Ids are not maintained by the operating system (and in most environments not even by the middleware layer) it is highly dependent on the system environment how these activity Ids are generated. Examples of how activity Ids can be added to a system are given below.

From the functional point of view for middleware-objects the same types of events are of interest as for language-level objects: those that allow to trace the sequence of objects invocations. In the previous section the language level tool accomplished that by generating an event in just one object's context, either in the invoking or in the invoked object. However, when considering distributed systems it turns out that it is not enough. In the time between leaving one object’s context and entering the other one the activity is “in-transit”. This delay cannot be assumed to be zero, as it was the case for local C++ invocations. It measures the delay introduced by the intermediate middleware and the network layer. In distributed applications it is often exactly this overhead that is of special interest, as it is an important parameter for object placement or the selection of the best copy of a replicated service. In order to measure this delay it becomes necessary to detect both events, when an activity is leaving of one object’s context and also when it is entering the other one. This happens twice for each synchronous object invocation: once for the request and once for the reply. Measuring the time between the two events is a truly distributed measurement as both events are possibly recorded on different nodes. It is important to realize that pure local "round-trip" measurements cannot provide the same information because typically the amount of required processing resources is not equally distributed between the two directions.

With the use of an object-oriented middleware layer invocations between two objects managed by this layer are redirected through the middleware. This simplifies the task of placing the sensors significantly. Like for the instrumentation of the operating system it is enough to augment the system with a few generic sensors at those points in the middleware that are passed by all invocations. Similar to the introduction of activity Ids, this requirement was not foreseen by the initial middleware standards. Again, the implementation is dependent on the

actual system environment. In the next section implementation alternatives for CORBA based middleware will be discussed an in the following case study on the integrated monitoring tool MagicZoom a concrete implementation also for DCOM will be described.

2.4.3 Instrumenting CORBA

The concept of an activity as some kind of a distributed thread is not implemented in a standard CORBA ORB. However, using the CORBA interceptors API allows to implement the concept of activities as an add-on.

Logically, an interceptor is a transformer interposed in the invocation (and response) path(s) between a client and a server object. Interceptors are intended as a generic mechanism for adding services to a CORBA-compliant object system in a portable manner. They are bound between client and server objects and they are derived from the interceptor interface defined in OMG IDL. CORBA defines two types of interceptors (see Figure 2.13):

- **Request-level interceptors**: are used to implement services, which may be required regardless of whether the client and server reside on the same host or not. They resemble the CORBA bridge mechanism in that they receive the request as a parameter, and subsequently re-invoke it using the Dynamic Invocation Interface (DII). The ORB core invokes each request-level interceptor via the `client_invoke` operation (at the client) or the `target_invoke` operation (at the server). Request-level interceptors are intended for services such as transaction management, access control, or replication. Services at this level process the request in some way. For example, they may transform the request into one or more lower-level invocations or make checks that the request is permitted. The request-level interceptors, after performing whatever action is needed re-invoke the (transformed) request using the CORBA Dynamic Invocation Interface. The interceptor is then stacked until the invocation completes, when it has an opportunity to perform further actions, taking into account the response before returning. Interceptors can find details of the request and the reply using the operations as defined in the Dynamic Skeleton interface of CORBA 2. This allows the interceptor to find e.g. the target object 1, operation name, context, parameters, and (when complete) the result.
- **Message-level interceptors**: When a cross-machine invocation is required, the ORB will transform the request into a message, which can be sent over the network. Here a second kind of interceptor interface is defined that manipulates messages. The ORB code invokes each message-level interceptor via the `send_message` operation (when sending a message, for example, the request at the client and the reply at the server) or the `receive_message` operation (when receiving a message). Both have a message as an argument. The interceptor generally transforms the message and

then invokes `send`. Request-level interceptors are intended for services such as e.g. encryption.

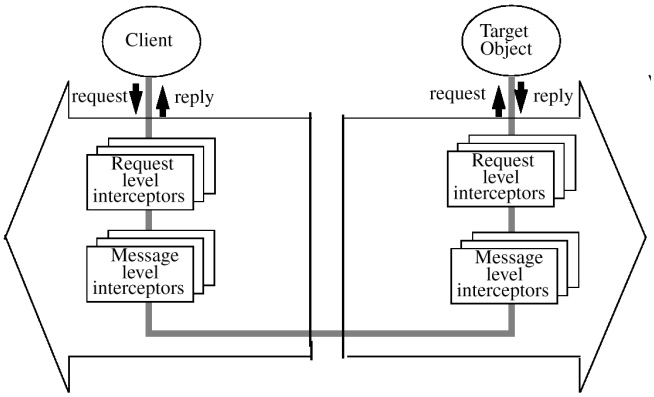


Figure 2.13: Interceptors in the CORBA 2 specification

For the purpose of instrumentation especially the request-level interceptors are of interest. They allow to introduce additional sensor code that will be executed every time an object is invoked. The method request is fully accessible including its parameters and its environment. The environment of a request is a set of named parameters that are not necessarily all specified in the interface description of the invoked object. Environment parameters that are not required by the invoked object are simply ignored there. This enables the interceptor to add extra parameters to the environment, which are afterwards marshaled and sent with the other parameters. The interceptor adds a unique activity Id (a random number) into the environment, if it cannot find a definition in the environment of the request. This is done every time a request or a reply leaves through an interceptor and the sensor code within the interceptor includes this information at the client and at the server side in to the event record that reports on the object invocation. This way the activity can be traced on its way through the distributed object space.

This basic instrumentation is generic for all objects implementation that are supported by the ORB. It has been first implemented using a predecessor of the CORBA interceptor interface, namely the filter mechanism of the IONA ORBIX ORB [Ion95, Ger97a]. ORBIX filters were introduced for the same purpose as interceptors later, but as they were an ORB-specific feature, their usage was limited to a small set of applications.

2.5 Case Study – MagicZoom

The previous sections described the concepts and implementation issues required when monitoring object-oriented real-time systems at the different architectural levels. This section now focuses on an integration of the techniques into one tool. The tool named MagicZoom allows to monitor the various levels at the same time, to combine this information in one graphical presentation, and to "zoom" into the systems abstractions as required for the analysis of timing related problems. It combines the high level abstraction provided by object-orientation with the low level system view: on the hand, an activity can be seen as sequence of (possibly nested) method invocations that walk through an abstract object space. On the other hand, an activity is executed by threads that are scheduled on CPUs, that are subject to interrupts, that are blocked or preempted, and that are finally needed to understand and explain the timing behavior of the application. MagicZoom monitors and visualizes the distributed execution path of an activity in terms of method invocations and returns while simultaneously revealing the status and execution times of the associated threads. MagicZoom is intended as a tool for developers that design, implement, and test distributed object-oriented real-time applications. However, while its graphical user interface addresses human users, a possible consumer of the provided information could be the runtime system itself as described in Section 3.

MagicZoom applies the monitoring concepts described in the previous sections. Therefore it is portable and applicable to any distributed object-oriented framework that supports (or can be extended to support) the concept of activities. Regarding implementation, the monitoring components are generic and portable. The instrumentation however (i.e., the application of event-generating sensors in the target system), is system dependent. Here the instrumentation for DCOM on Windows NT is described [Moc00]. While DCOM has been specified as a platform independent standard, and implementations for Unix, Linux, and other operating systems are available, the premier platform for DCOM is Windows NT. It was the first platform providing DCOM support, and since version 4.0, it contains the DCOM runtime-libraries in its standard distribution.

As stated in [Cus93] "Microsoft® Windows NT™ Workstation is not a hard real-time operating system. Rather, it is a general-purpose operating system that has the capability to provide very fast response times, but is not as deterministic as a hard real-time system". Also, there is no "Real-Time DCOM" specification available. Why has DCOM/Windows NT been chosen as primary target for a monitor that aims at providing service to real-time applications?

1. Because DCOM is the base for many commercial real-time projects. DCOM is established on the market as widely used product and there is a growing interest in DCOM in the field of industrial process control. Industrial automation systems are getting more connected with higher level processes, the use of off-the-shelf components and hardware independent software standards

becomes a predominant factor for the marketing. All this makes DCOM an ideal candidate as basis for future developments in that field.

2. Windows NT itself comes with many valuable mechanisms for soft real-time applications, like e.g. multimedia applications or online stock-trading. It provides preemptive multi-threading with special real-time priorities, a memory-management that offers page-locking, and a kernel that is optimized for minimizing interrupt latency by the use of deferred procedure calls. And, last not least, the Win32 API provided by Windows NT acts as de-facto industrial programming standard.

3. Real-time applications on top of a not completely predictable system-platform can benefit even more from any kind of time-awareness. As there is inherently no static worst case timing analysis, there is an even bigger need for information on the actual behavior at runtime.

4. Any system that uses an object-oriented middleware layer uses a standard non-real-time communication layer (like TCP/IP), looses its overall predictability. Again, in such a standard distributed environment there is a need for information on the actual behavior at runtime.

Since the applied system abstractions (activity, objects, invocations at the higher level, processes, threads, etc. at the lower level) are identical, the monitoring concept is applicable to Real-Time CORBA, too. In fact, activity handling in MagicZoom uses the same approach as the CORBA scheduling service of using names (strings) for identifying activities and objects.

2.5.1 Monitoring with MagicZoom

A user of MagicZoom controls a monitoring session from the central monitoring console with its graphical user interface (see Figure 2.14). This monitoring console is typically hosted on a dedicated node that itself is not involved in the monitored distributed activity. From this interface an experimenter initializes all involved nodes, i.e. he/she selects the event-types of interest and starts the event recording on these nodes. Now the distributed object-oriented program can be run. Each node records its local event stream, i.e. at least all object-related events. If explicitly selected by the experimenter, also thread switches, interrupts, and user-level events will be traced. An observation ends by stopping the event recording from the MagicZoom user interface. Automatically, the event streams from all nodes are transferred to the central monitoring console.

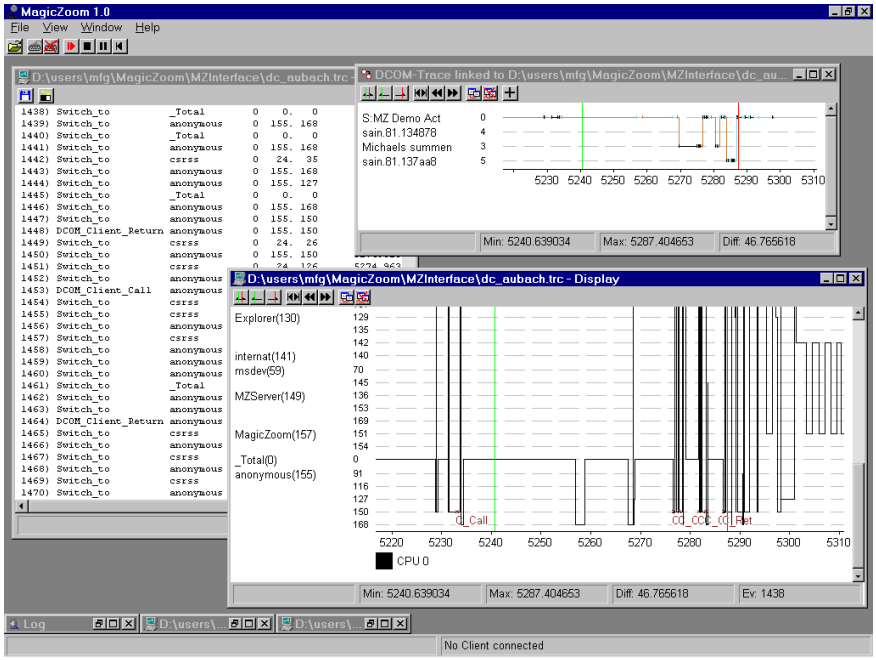


Figure 2.14: The MagicZoom user interface

In order to analyze the walk of an activity through the distributed object space, the experimenter opens a view called “DCOM-Trace” at the central monitoring console. An initially empty window appears and the experimenter gets a selection of all observed activities from which he/she can select the activities of interest. These activities are presented in a Gantt-chart (see Figure 2.15, the Gantt-chart at the bottom).

The y-axis lists all objects visited by the selected activities and their called methods. Objects are identified by their user-defined names or, if such a name is not available, a system-generated name is used. Methods are simply named by their index in the virtual function table (as method names are only known to the compiler, not to the DCOM runtime system, it would require an additional name service to provide symbolic names here). Since the starting point of an activity is not canonically linked to an object, each activity is associated with a new pseudo-object called “S:<activity-name>”. It represents the activities initial method, similar to a “main()” in C++. On the x-axis, the global time from the synchronized clocks of the system is given. Each selected activity is represented by a line in the chart that walks through the different methods of the system’s objects (like a thread with a call stack in the local case). Areas of the line that are separated by small vertical markers depict different states of the thread currently executing on behalf of the activity. The thread can be executing in normal mode,

it can be waiting, executing an interrupt, or the CPU is idle (in the graphical interface these different states are visualized by different colors). For getting further information on the local behavior of an activity, for instance in order to find out by which other thread the thread of the activity has been preempted, an separate different view showing all threads on a node can be opened.

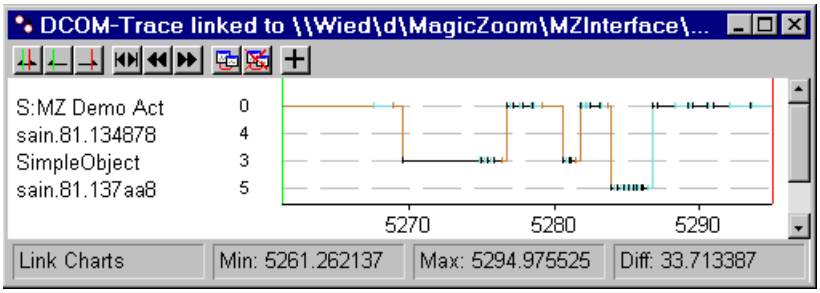


Figure 2.15: The DCOM-Trace view of MagicZoom

Figure 2.15 shows the visualization of a very simple example of a DCOM activity. It contains one selected activity, four objects, and one method(-number) per object. One of the objects is the pseudo-object "S:MZ Demo Act" that represents the initial method of the selected activity (since the user-defined name "MZ Demo Activity" has been assigned to the thread). Two of the three remaining objects have their system-generated names ("sain.81.134878" and "sain.81.137aa8") and one has the user-defined name "SimpleObject". The two objects with system-generated names are those of the class factory (DCOM internal helper objects) that are only required for constructing and destructing the "SimpleObject" object, which finally provides the user-defined functionality that the application wants to invoke. The chart further shows the invocation of method 3 of the "SimpleObject" object and the call to the final destruction (method 5 of the class factory object). In the given screenshot the view on the activity trace has been zoomed onto a short time-interval of overall about 100 ms. The invocation of method 3 of the "SimpleObject" object that contains virtually no user-code already takes about 7 ms on a pair of a Pentium 233 MHz (client) and a Pentium 133 MHz (server) machine with no other load.

2.5.2 The Design of MagicZoom

Figure 2.16 depicts the architecture of the components on one node monitored by MagicZoom. Basically, it comprises three components similar to those of JewelNT, the instrumentation, a remote communication infrastructure (both one instance per monitored node), and a central monitoring console with its graphical user interface. The instrumentation includes the kernel driver and libraries. The kernel driver augments the kernel with the sensors for detecting the operating system level event and maintains the per-node event buffer. The libraries imple-

ment the instrumentation of the DCOM middleware layer as described below. The remote communication infrastructure is responsible for controlling the monitoring and for retrieving the event traces from the event buffer, for buffering the data and for transferring it to the central monitoring console. This console collects event traces from the nodes of the distributed system, synchronizes them, and merges them into a global view of the system. Finally, it is responsible for computing and displaying the activity traces.

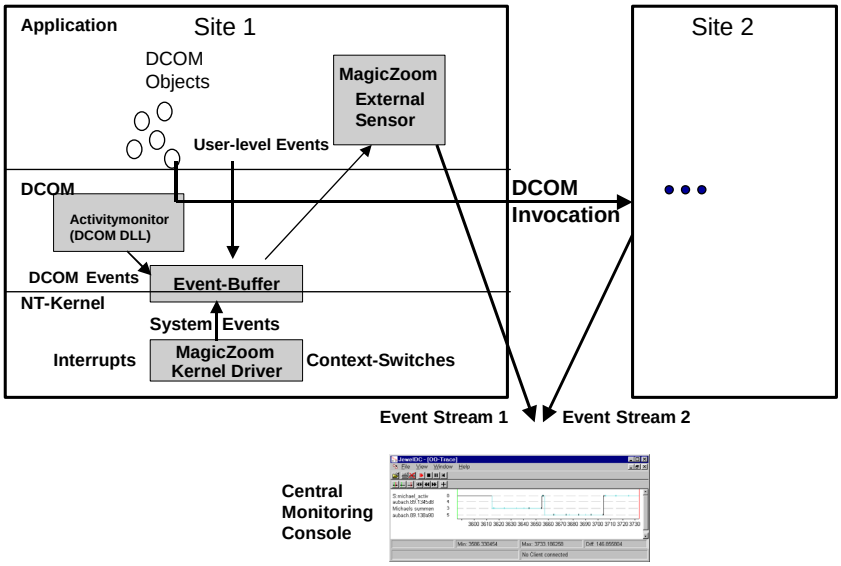


Figure 2.16: Components of MagicZoom

User-Level API

The high level abstractions on which the monitoring of the distributed object-oriented system with MagicZoom is based are objects and activities. For identifying instances of objects and activities, concepts of object identifiers and activity identifiers are needed. This is especially necessary when system behavior is monitored: monitored information is presented with object identifiers and identifiers that allow the experimenter to associate the monitoring information to the respective system entities. In order to make a generic design independent of the concrete representation of object and activities in a specific system, it is possible for the user to assign names (user-defined strings) to objects. These names are then displayed in the monitoring interface.

Figure 2.17 depicts the programming interface of the MagicZoom activity monitor. The function `AM_set_object_name` associates the object pointed to by the parameter `obj_ptr` with name given by the parameter `object_name`.

```
AM_API void AM_set_object_name(
     void *obj_ptr,
     char *object_name);

AM_API void AM_set_activity_name(
     char *activity_name);

AM_API set_user_event(
     int user_param);
```

Figure 2.17: Programming interface of MagicZoom

A similar solution is applied for identifying activities. In terms of operating systems entities, an activity is represented by an initial thread executing a program in some process. Secondary threads in other (local or remote) processes are temporarily added to the activity when executing method invocations on behalf of the initial thread (or on behalf another secondary thread that currently belongs to the activity). Note that secondary threads can (sequentially) work on behalf of different activities. In order not to overwhelm the user with these low-level system issues, which in addition can change from run to run, the activity monitor allows to associate activities with user-defined names. This is the purpose of the function `AM_set_activity_name` in Figure 2.17 that has to be called by the initial thread of an activity and associates the name given by the parameter `activity_name` with that activity.

Event Types

The monitor distinguishes three event categories:

1. the object-related events from the abstraction level of the distributed object-oriented framework occur whenever an activity invokes (or returns) from an object,
2. low-level system events such as thread switches and interrupts that relate to the scheduling behavior on a local node
3. user-level events in the application code. These include events generated by the *mc4p* automatic instrumentation tool.

The object-related events allow tracing an activity through different threads, nodes, objects, and methods. They are defined as follows:

1. Outgoing call: a thread *t* on node *n* executing on behalf of an activity *A* invokes some object with call-id *i*. The call-id is a unique number generated by the instrumentation that allows matching outgoing and incoming calls/replies.
2. Incoming call: a thread *t'* on node *n* starts executing a method *m* in an object *O* on behalf of an activity *A* with call-id *i*.

3. Outgoing reply: a thread *t'* on node *n* ends executing a method *m* in an object *O* on behalf on an activity *A* with call-id *i'*.

4. Incoming reply: a thread *t* on node *n* executing on behalf of an activity *A* receives a reply from some object with call-id *i'*.

The implementation details of how the object-related events are retrieved from the DCOM runtime system are explained below. Low-level system events are retrieved from the operating system kernel by instrumenting that kernel (as described in section 2.2). Finally, user-defined events are indicated with the function `set_user_event` in Figure 2.17. The application programmer can set the parameter `user_param` for identifying this event later on. The user-defined event allows the application programmer to mark specific points of interest in the execution path, e.g., code sections of interest within a method. Mc4p provides a subclass of *_instrumented_class* that uses `set_user_event` to record its events.

Instrumentation of the DCOM-Middleware Layer

The concepts of object and activity identifiers are the basis for gathering the object-related events from the system. For the convenience of the user, MagicZoom allows to associate user-defined names with objects and activities for monitoring purposes. However, a system level representation of object identifiers and activity identifiers is needed, too. Since DCOM does not directly support the notion of globally unique object identifiers, a 3-tuple <server process site id, server process id, virtual address of the object in the server process> is used as system level unique identifier for an object. The virtual address of an object is the address of the object's implementation of the IUnknown interface, the basic interface that every DCOM object must implement.

Let us now consider how the DCOM system is instrumented in order to detect the object-related events. This will also reveal how the notions of activity and activity identifiers are introduced to DCOM. Invoked objects can reside either in the same process or in another (either local or remote) process - it is called "out-of-process". Invocations of out-of-process objects imply changing the thread that is active on behalf of the activity. Special care has to be taken that the monitoring system can keep track of the invocation under the activity id for ensuring that the involved thread are monitored as acting on behalf of the same activity.

An invocation of an out-of-process object is implemented by a remote procedure call (RPC) that is executed between the stub object and the actual object. The RPC protocol used by DCOM basically is DCE RPC [Loc94] with the extension that some extra information is piggybacked on the RPC call/reply messages.

Since its latest release DCOM defines in its API the `ChannelHook` interface that allows defining and extracting the extra information piggybacked to the RPC call/reply messages [Edd99]. By default, DCOM adds a `CausalityIdenti-`

fier to each call and reply. This identifier remains the same whenever a call is executed on behalf of another call. It supports tracing of activities since the CausalityIdentifier can be used as internal identifier for the activity. The ChannelHook interface provides four functions ClientFillBuffer, ServerNotify, ServerFillBuffer, and ClientNotify that are called by DCOM when an RPC is made, an RPC comes in, an RPC reply is sent, and an RPC reply is received, respectively. These functions are called whenever the object-related events (see section 3.1) are to be generated. Hence, the required instrumentation of DCOM can be achieved by implementing the ChannelHook interface. As an example, Figure 2.18 shows the implementation of the ServerNotify function that generates the "incoming call" event.

```
// Called in server just before a method is invoked

void CChannelHook::ServerNotify(
               REFGUID uExtent,
               REFIID riid,
               ULONG cbDataSize,
               void* pDataBuffer,
               DWORD lDataRep)
{
   SChannelHookCallInfo *p_callinfo;

   if(uExtent == EXTENTID_MyHook &&
      lDataRep == NDR_LOCAL_DATA_REPRESENTATION)
   {
        p_callinfo = (SChannelHookCallInfo *)&riid;
        MYHOOK_THIS* data = (MYHOOK_THIS*)pDataBuffer;

        MagicZoomNoticeOO(
               MZ_EV_OO_SERVER_CALL,            // Event-Id
               p_callinfo->uCausality,          // Activity-Id
               object_id(p_callinfo->pObject), // Object-Id
               p_callinfo->iMethod,             // Method-Id
               *data);
   }
}
```

Figure 2.18: Generating the incoming call event with a ChannelHook

The code shown in Figure 2.18 extracts the relevant extra information from the incoming RPC and stores it in the local event buffer of the monitoring system. The helper function object_id returns a pointer to the object's IUnknown interface. The other functions of the ChannelHook interface are implemented in a similar way, yielding a complete instrumentation of the object-related events in DCOM for out-of-process invocations, including the information on the overhead introduced by the RPC system.

The current implementation of the DCOM instrumentation does not generate events for invocations that do not cross process boundaries. These calls are executed just like simple C++ invocation and information about them is not neces-

sary for keeping track of activities. They are instrumented at source code level using the mc4p tool.

Tracing Activities

The object-related events are recorded in the different traces from the various machines involved in the object-oriented computation. Now, how is information of distributed activities extracted out of this data? The different traces are first transferred to the central monitoring station and then projected onto the same global time axis. This adjustment of the local time stamps is done as described above for JewelNT in subsection 2.2.2. In the next step, all recorded event streams are scanned for object-related events. The collected activity-ids from these events together with the involved object-ids are stored in a separate look-up table that is used for preparing the visualizing chart. The table stores any user-defined name that the application has assigned to one of these instances as well as the thread that contains the first occurrence of an activity.

If now the experimenter selects a certain activity *A* for display in the O-O Trace view, the algorithm depicted in Figure 2.19 collects the activity trace information S_A for this activity from the various event streams. It is executed once for each monitored activity. Basically it creates the activity trace information S_A by following the thread between the object invocations, as it is the implicit vehicle of an activity, and following the explicit activity-ids across process and machine boundaries. S_A contains all events from the different nodes that happened during the time the activity was active on that node. The visualization uses this information to present the activity, its execution status and the objects it has invoked. Note that in step 7) a special error indicator is set if an activity invokes an object on a node that has not been monitored. In this case, the visualization shows a dashed line until the call returns, which indicates this condition to the experimenter.

1) *t* (the current thread) *:= NULL*
2) S_c (the current event stream) := *event stream that contains the first occurrence of A*
3) S_A (the event stream for activity *A*) *:= empty*
4) skip all events from S_c until the first occurrence of a DCOM-event *e* with activity-id$_e$ equals *A*
5) *t* := thread-id$_e$
6) copy next event *e* from S_c to S_A
7) if thread-id$_e$ = *t* and *e* is an outgoing invocation with call-id$_e$ = *i* find other event Streams *S'* for an event *e'* where *e'* is an incoming invocation with the activity-id$_{e'}$ equals *A* and call-id$_{e'}$ = *i*; if found skip all events before *e'* in *S'*; $S_c = S'$; *t* := thread-id$_{e'}$; continue with 6); if not

found (the node hosting the object has not been monitored) copy a special event e_{error} to S_A and skip all events before the next incoming reply event *e''* in S_c with activity-id$_{e''}$ equals *A*

8) if thread-id$_e$ = *t* and *e* is an outgoing reply with call-id$_e$ = *i* find other event Streams *S'* for an event *e'* where *e'* is an incoming reply with the activity-id$_{e'}$ equals *A* and call-id$_{e'}$ = *i*; if found skip all events before *e'* in *S'*; $S_c = S'$; *t* := thread-id$_{e'}$; continue with 6)

9) if S_c is not empty continue with 6)

10) end

Figure 2.19: Algorithm for constructing a global activity trace

2.5.3 Summary

The MagicZoom allows for the tracing of activities in object-oriented applications while simultaneously revealing and visualizing the CPU scheduling of the threads that execute on behalf of the activities. Being a valuable tool for the development of distributed object-oriented real-time applications, it also complements the Real-Time CORBA approach that specifies scheduling support for distributed activities.

MagicZoom has been implemented in the context of DCOM. Although lacking a clear concept of global object identifiers, DCOM has turned out to be a suitable implementation vehicle. Especially the concept of ChannelHooks has directly supported the instrumentation of DCOM and allowed for the introduction of the concept of distributed activities into DCOM. DCOM is of major interest in industrial process control systems and MagicZoom has been applied to performance and real-time related problems in a DCOM-based distributed factory automation system.

3 Real-Time Systems

The previous chapter discussed time-aware systems that use monitoring to get information about their own timing. While these systems have the system-supported infrastructure to react on time-related problems and to adapt to an actual timing behavior, they do not yet fulfill the basic requirement of real-time systems. The time-awareness in itself does not provide any guarantees that a specified timing behavior is accomplished. It does not include any mechanisms for exploiting the acquired knowledge about its own timing. It is still up to the application or to the designer of the system to take advantage of the newly gained knowledge and to implement a strategy for achieving the final goal of any real-time system, namely to provide guaranteeing timely correct behavior. This chapter discusses how time-awareness can be utilized to accomplish these guarantees event in an environment as described in the introduction where real-time and non-real-time objects interact in one application.

Section 3.1 of this chapter extends the monitoring techniques presented in the previous chapter towards a system architecture that can provide these guarantees. It first describes the general problems with obtaining timing bounds. It argues that the abstraction of worst case timing assumptions, as adopted by the vast majority of previous publications on time-critical computing, leads to a very limited view on real-time systems. This limited view excludes the efficient use of modern hardware as well as many of the well-known features of object-oriented programming. In order to broaden the scope of real-time systems, a task-classification is presented that introduces a category of soft tasks. These soft tasks allow for a tradeoff between timeliness and functionality. The TAFT-Scheduling approach is presented that implements a scheduling that is aware of this tradeoff. TAFT-Scheduling relies on up-to-date timing analysis provided by a monitoring component. In section 3.2 the concept of Expected Case Execution Times (*ECET*s), the basis for providing timing estimates for the TAFT-Scheduler, is introduced. The remainder of this section discusses algorithms for an efficient analysis and prediction of ECETs, extends the concept towards an early detection mechanism for probable timing faults, and describes the tradeoff between the achieved granularity of ECET analysis and the required resources. The implementation architecture for the TAFT-Scheduler in general and especially the components of the online monitor with ECET-analysis are presented in section 3.3. Section 3.4 concludes the discussion of the tools by providing performance figures that prove the feasibility of the concept. Finally, in section 3.5 the general applicability of the presented algorithms and implementation structures is illustrated by a study that applies the monitoring and event processing components to a slightly different problem in object-oriented real-time computing, namely the online checking of formal timing constraints.

3.1 Providing Timing Guarantees

In a real-time system, it is mainly the job of the scheduler to guarantee timely correct behavior. It assigns the required resources to the specified tasks and determines whether these will meet their specified timing in all possible executions of the complete systems. *The required resources* certainly include the computing power, i.e. the CPU, but may include also network bandwidth, memory, access to critical regions, and any kind of physical resources attached to the system. In general the complexity of a scheduler's job grows exponentially with the number of tasks. In the past, a lot of work on scheduling has been done and the most relevant results for current real-time systems are based on more or less rigid constraints on considered the tasks and resources [But97]. Often a very limited set of resources is considered and the most commonly used scheduling algorithms, e.g. RM (Rate Monotonic) and EDF (Earliest Deadline First), consider the CPU-resource only. All these scheduling algorithms have in common that they need a set of attributes of the tasks to be scheduled in order to perform a schedulability analysis, i.e. a test that determines whether a given set of tasks can fulfill its timing requirements or not. The attributes comprise at least the period (in case of periodic tasks), the release time, and the deadline of a task. The values for these attributes can be extracted from the specification of the system. One other important attribute cannot be determined so easily: the execution time.

3.1.1 Worst Case Execution Times

As the actual execution time depends on the current state of the system that is hardly known in advance, schedulers usually base their planning on the *worst-case execution times* (*WCET*) of the tasks. The *WCET* is the maximum execution time that an arbitrary instance of the task ever needs until completion.

More formally: Let *t* be an instance of a periodic task *T*.

$ET_t :=$ The CPU-time instance *t* needs until completion in the given environment.

The CPU-time includes only the time when the task is assigned to run on the CPU. It does not account for any queuing-time or for any overhead introduced by the executing environment. With this definition the *WCET* of a task *T* is given by:

$$WCET_T := Max_{t \in T}\{ET_t\}$$

While the *WCET* of a task is usually also given in the specification, there is no constructive approach that can generate code for a task with a given functionality and that can guarantee a fixed *WCET*. Therefore, an iterative approach is taken. The code is implemented and then it is determined whether this implementation can meet the requirements. If not, either the specification of the *WCET* is adapted

or the implementation or the execution environment is modified in order to match specification and reality.

This approach needs a reliable method to determine the *WCET* of a task. The straightforward approach would be the analysis of the program code that searches for the longest possible path in a task's execution and accounts for its execution time. However, in real life this imposes more and more problems and most programming environments do not provide the tools for this kind of analysis. This is because:

- A programming language that can be statically analyzed for *WCET*s necessarily has to impose severe restrictions on the expressiveness of its statements. Loops must be bounded and recursion must be restricted in depth. Thus, also dynamic data-structures are restricted. Trees or lists must have a fixed depth and length and it is impossible to manage an object-space of previously unknown size. In dynamic systems that schedule tasks dynamically at runtime, it is possible to give a function for a task that computes its *WCET* depending on the current system status. However, usually it requires additional effort from the programmer to provide the knowledge for this function.

- Another problem is that *WCET*s are highly system specific. The *WCET* of a program does not only depend on its logical structure but also on the compiler, the processor, and the system environment. Thus, while there are some approaches to compute *WCET*s with only the source code and some system knowledge as input, most attempts to determine the *WCET* rely on the detailed knowledge of the processor, the source code, and the compiled object code. But this is still not sufficient. The characteristics of a single system, like its clock speed, its bus architecture, its memory access behavior, and its DMA controller, have significant and often non-linear impact on the timing of a program. Thus, it is impossible to argue about *WCET*s without a specific machine in mind.

- Another problem is the increasing complexity of today's hardware architectures. Much of the basic work on *WCET* analysis in the past has been done on M68000 machines, which had nearly constant execution times per instruction. This means, from the timing point of view a pretty simple model. However, all modern processors make intensive use of at least two levels of caches (instructions and data), internal pipelining and parallelism, branch prediction, and speculative execution. A model that tries to cover all these features can become about as complex as the hardware itself. An model that ignores only one of these architectural features can easily lead to a massive overestimation of the actual WCET (e.g. 275% for data caching as shown in [Kim99]).

- Even with a perfect model of the executing machine, it becomes more and more difficult to give reasonable bounds for the *WCET* of a piece of code.

The speedup of processors in the last years does not mainly result from faster transistors, but from a better usage of parallelism and locality. The current state of the machine has a big impact on the execution time of the next instruction. However, it is hard to determine the dynamic state of the computation in advance during static analysis. Modern processors are not built for constant execution times, but for a high probability of a maximum execution throughput. This and the two prior issues lead to the general observations that:

1. It is really hard to determine how long a single instruction takes in a certain execution context and
2. the difference in the execution time between the worst case and the average case may be in the order of 10.

The first observation means that it is even more difficult to build a WCET analyzer than it was ten years ago and the second observation tells that even a good tool will provide WCET figures that are far apart from the realistic execution times of a running application. Drastically spoken, this means that real-time systems that rely only on *WCET*s will not benefit a lot from the current development of new processors.

- Finally, object-orientation and, based on it, component based programming impose another major problem on *WCET* analysis. Using existing (or even third-party) components in real-time applications is difficult because the timing behavior of the components must be known. While implementation hiding isolates module implementations and eases their integration as it allows matching interfaces easily, it ignores the fact that for execution time analysis knowledge about the implementation at the lowest possible level is required. Polymorphism and late binding [Boo91] (*virtual* methods in C++) impose another problem. As in an object-oriented environment the code that is actually being executed can be determined at runtime, *WCET* analysis has to take into account all possible implementations.

If it so hard to get reasonable bounds for the *WCET* of tasks, especially in object-oriented systems, what can be done? How can the benefits of object-oriented software development and new processor architectures be transferred into the real-time domain? The answer heavily depends on the type of application and especially on the class of tasks that are considered.

3.1.2 Task-Classification

Classically, real-time tasks have been divided into “hard” and “soft” tasks. However, to capture both the functional and timing behavior, this scheme is insufficient. In [Kai99] a classification of real-time tasks is given that will be used in the following to identify those tasks that are candidates for a real-time scheduling beyond the restrictions of a full *WCET* analysis:

1. **Hard Tasks:** The term *hard task* is well known in real-time literature. It denotes all those tasks, where it is mission-critical that these tasks are executed and those they meet their timing requirements. Any timing-fault of a hard task will lead to a non-acceptable failure of the whole system, depending on the embedding system with possibly catastrophic consequences. The timely correct execution of hard tasks must be guaranteed by the real-time system and when executed they must not fail. A typical example of a hard task is the brake-by-wire control in an automotive.

2. **Essential Tasks:** Similar to hard tasks *essential tasks* also must not violate their timing requirements. Essential tasks are generating the computational progress of the application and once they are started, they have to be treated like hard tasks. However, in contrast to hard tasks, there is some flexibility in *when* they are executed. The real-time system can decide to delay the execution of an essential task until it can guarantee the resources for a successful execution. Usually, there are still requirements on the maximum allowed delay or the overall rate of executions. The notion of essential tasks has been first introduced by Stankovic and Ramamritham in the Spring kernel [Sta89]. A classic example of an essential task is the landing-control of an aircraft. It has to be executed in the near future and once the task (or a set of cooperating tasks) has been started it has to be completed successfully.

3. **Soft Tasks:** The term *soft task* is also often used in real-time literature. However, its definition is usually quite fuzzy. In many papers the term soft task only means *not* a hard task, including all kinds of tasks ranging from essential to non-real-time. A common understanding is that a soft task might fail to deliver the desired functionality within the specified time. In order to distinguish the functional and the timing dimension, in [Kai99] a division into "soft" and "best-effort" tasks is proposed. With this definition, a "soft task" is a task that has to meet its timing specification, but it has the option to provide functional degraded behavior. An important constraint remains: the result of a soft task must not violate the safety constraints of the complete system, i.e. it is allowed to provide no computational progress, but it must not leave the system is an erroneous state. When looking closer to tasks that are considered to be hard real-time, it turns out that many of them an actually soft tasks in the sense of this definition. Often, it is acceptable and also more cost-effective to use a smaller sized system that might fail in e.g. one percent of its task executions than to use a huge over-sized system that handles even the worst case. This kind of sporadic degradation of functionality might be tolerable if the system still maintains a basic level of functionality and safety by guaranteeing a hard real-time core. Typical examples of soft tasks are all kinds of "any-time" algorithms that provide increasingly better results for the same problem, depending on the amount of execution time they can use (like e.g. a chess program). Also media-streaming applications often reveal soft task behavior as they either drop images of a video-stream or reduce the reso-

lution of the audio or video signal in response to insufficient computing or networking resources.

4. **Best-effort Tasks:** Finally, *best-effort* tasks are those tasks that always provide full functionality, but are allowed to execute beyond their optimal completion time (i.e. their "deadline"). The real-time system will try its best to assign enough resources to these tasks to make them meet their timing requirement, while it does not provide guarantees. The idea of assigning a value function [Jen85] to each task that expresses the value of a completed task at a certain point in time can be applied to express the overall value of a run of a system of best-effort tasks. Typically, e.g. all kinds of timing requirements that are expressed for (Web-based) user-interfaces of business-applications can be categorized as being best-effort tasks.

All tasks that have no explicit timing specification and that should just be executed “as fast as possible” are “non-real- time” tasks and will be not considered in this context.

Regarding this classification hard and essential tasks are not good candidates for diminishing the requirements on WCET analysis. As these task have to provide full functionality in time and a (timing) fault of these tasks has possibly catastrophic consequences, there is no alternative to a full schedulability analysis (including *WCET*s). On the other hand, best-effort tasks don't need the strict WCET attribute. For a successful scheduling of best-effort tasks it is enough for the scheduler to get stochastic values for their execution times. Sporadic failures in meeting their optimal completion times will not reduce the overall value of the systems significantly. Only permanent timing errors will degrade it beyond an acceptable quantity.

Now, what about soft tasks, the remaining category? Does a scheduler need the *WCET*s of these tasks in order to guarantee the safety of the system? Not necessarily. It can use probabilistic timing assumptions for scheduling the functional part and find a way to deal in time with the situation when a really bad case happens! It is not required that each execution of a task is successful. It only has to be guaranteed that the effects of a termination with degraded functionality of one instance do not cripple the systems overall state nor the timing of the remaining tasks. Also, the scheduler should ensure a considerable amount of successful executions of task instances. Otherwise the computational progress of the system is in danger. Once the system can assure this, the soft task concept enables a large quantity of currently not real-time capable code to work in a predictable manner under timing constraints.

In order to implement this idea three issues have to be resolved:

1. The tasks have to be organized in a manner that allows to separate the hard real-time core (that ensures a consistent system state) from the functional part,

2. the scheduler has to be aware of this separation and schedule them accordingly, and
3. good timing estimates are required that enable the scheduler to spend enough resources on the functional parts in order to ensure computational progress.

The *Time-Aware Fault-Tolerant* (TAFT) scheduling approach, presented in section 3.1.4, addresses these issues and, thus, is capable of handling soft tasks in an adequate manner. It uses time-awareness to estimate the actual resource requirements of the functional parts. In order to handle this quantity, it uses a newly introduced property of a task, its *Expected Case Execution Time*. The notion of this property is formalized in the following section.

3.1.3 Expected Case Execution Times

Roughly speaking, the *Expected Case Execution Time* (ECET) is a measure for the time that instances of a task need in most cases for a successful completion.

Let t be an instance of a periodic task T.

$ECET_{t,p} :=$ the CPU-time that has to be assigned to instance t in order get a probability of p that t is completed.

Again, the CPU-time includes only the time when the task is assigned to run on the CPU. Similar to the definition of $WCET_T$ also the $ECET_{T,p}$ of a task T can be defined as:

$$ECET_{T,p} := Max_{t \in T}\{ECET_{t,p}\}$$

This property of a task denotes the time the scheduler has to assign to each instance of T in order to achieve an overall probability of successful completions of at least p. Note that, $WCET_T$ is an upper bound for $ECET_{T,p}$ as even for p=1 all instances of T will complete within $WCET_T$.

In contrast to $WCET_T$ and $ECET_{T,p}$, which are static properties of a Task, $ECET_{t,p}$ is dynamic. It is not only dependent on the code to be executed by the task T (and its possible set of parameters) but also on the system's state and the environment at the time T is executed. This means that the $ECET_{T,p}$ may change over the live-time of the system and, if the scheduler can handle dynamic changes in the task descriptions, it is exactly the quantity the scheduler needs. As it describes the requirements of the a single instance, the scheduler can adapt to the current behavior of the system's tasks. In general, the task of the scheduler will become too complex if it has to use new timing estimations for each instance. However, the "locality" of *ECET*s can be expected to be high. This means, in many cases the $ECET_{t,p}$ will change marginally from one instance of T to the following. Especially, this is true for all T that have no or only a small data-dependency of their execution times. This is also true for tasks that operate on

dynamic data where the amount of data is increasing or decreasing slowly. The assumption still holds for many cases of changes in the environment, e.g. varying physical properties of sensors or actuators (e.g. caused by their increasing age) or changes in the overall load on the system. In case of transient errors (like e.g. message loss in the network and a repeated computation) this is surely not the case, but these cases can be regarded as equally distributed over all executions. The case that the $ECET_{t,p}$ is monotonously increasing is pathological because either the task is unusable in a real-time system as it will exceed all bounds or the value will converge towards a fixed time.

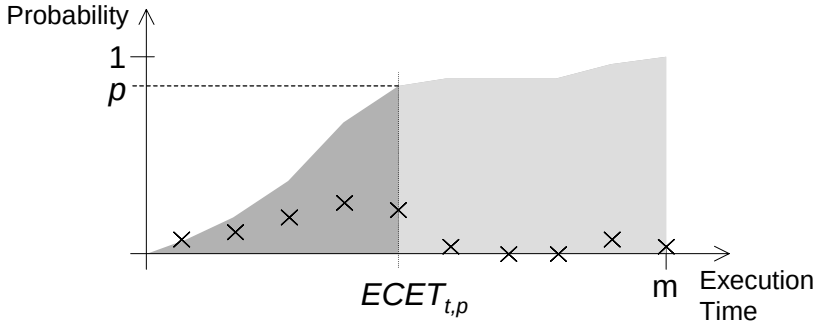
The $ECET_{t,p}$ can be derived from a discrete probability density function. Consider a density function f_t for an instance t of task T

$$f_t : \{1,2,...,m\} \rightarrow [0,1], where \sum_{x=1}^{m} f_t(x) = 1$$

where $m = WCET_T$, (given that T has some $WCET$) with the interpretation that $f_t(x)$ is the probability that the execution time of the task instance is exactly the discrete time x. From this function $ECET_{t,p}$ can be computed by

$$ECET_{t,p} = Min\left(q \in \{1,2,...,m\} \mid \sum_{r=1}^{q} f(r) \geq p \right)$$

This function is depicted in Figure 3.1. The points mark the values of the discrete probabilistic density function f_t and the shaded areas represent the sum of the probabilities up to that execution time (i.e. the probability distribution). This sum exceeds the given probability p for the requested $ECET_{t,p}$ on the execution time-axis (the dark shaded area). This is the p-quantil of the distribution.



.

Figure 3.1: Computing the $ECET_{t,p}$ from a probabilistic density function

However, $ECET_{t,p}$ is not yet a useable quantity for a real implementation of a scheduler as it is a probabilistic quantity that is not available in a concrete sys-

tem. The transition from a probabilistic to a statistic quantity is required. In order to estimate $ECET_{t,p}$ for the next instance t of task T it should be a good approximation to look at the most recent executions of instances of T. This leads to the definition of $ECET_{t,k,n}$. It denotes the time in which a specific amount of previous instances of the same task has succeeded.

$ECET_{t,k,n}$:= The minimal execution time that was needed to successfully complete at least k out of the last n executions of instances of a task T before t.

If the assumption of locality is true, then $ECET_{t,k,n}$ is a fairly good approximation for $ECET_{t,p}$ with $p=k/n$. $ECET_{t,k,n}$ can be computed from the statistical density function in the same way as $ECET_{t,p}$ has been derived above from the probabilistic density function. All that is required is the density function of the n most recent executions. This function can be obtained easily from a real system by online monitoring. This means, the $ECET_{t,k,n}$ is a good estimation for scheduling decisions of soft tasks and it can be made available in a real system.

3.1.4 TAFT Scheduling

The scheduler is the crucial component of a real-time system. It has to assign the resources to the tasks such that their timing requirements are met. Given an environment, where no reliable bounds for the *WCET* of tasks are available, but there is an understanding of soft tasks as described in subsection 3.1.2 and support for determining *ECET*s as described in the previous subsection. What is needed is a scheduling component that is capable of handling uncertain, possibly wrong timing parameters due to their estimated nature and dynamically changing behavior. This can be done by trading optimal functionality for timeliness, a well-known strategy in fault tolerance. Fault-tolerant mechanisms can be used to handle timing faults such that deadlines are still met. A timing fault occurs whenever the actual execution time of a task differs from its estimation. This idea lead to a system that is able to adapt online to changing timing parameters of the executed tasks: TAFT.

The *Time-Aware Fault-Tolerant* (TAFT) scheduling system consists of two major components: the Fault-Tolerant (FT) scheduler that enforces predictability and the Time-Awareness (TA) components, i.e. the monitor that is responsible for providing the required *ECET*s. The first ideas, the design of the TAFT scheduler, as well as reports on its usage in a concrete application example have been published in [Ger96a, Net97a, Net97b, Net98, Net01].

Fault-Tolerance

FT-Scheduling is based on the notion of a *TaskPair* (TP). Each task is designed as a *TaskPair* (TP). A TP constitutes a *MainPart* (MP) and the *ExceptionPart* (EP) [Str95]. From the scheduler's point of view, both parts are treated as sepa-

rate scheduling entities having their individual timing parameters. The minimal functionality of the EP is to ensure that the respective TP leaves

- the controlled application in a fail-safe state and
- the controlling system in a consistent state.

This reflects the above-mentioned fault-tolerance aspect. The deadline of the EP is identical with the one of the whole TP (see Figure 3.2). Its timely completion is guaranteed by the scheduler by explicitly reserving the necessary resources (e.g. the complete CPU time needed). Thus, its timing parameter ”execution time” can be interpreted as a *WCET*. This is reasonable because the EP usually comprises only a few, a priori determined system operations. In contrast to exception handling mechanisms known for other real-time programming environments, the scheduler guarantees the completion of the EP *before* the deadline of the TP, not, as usual, only as a best-effort task after the MP failed. Scheduling algorithms for a dynamic planning of TaskPairs are described in [Str95] and in [Kri97]. The latter calls the same concept “Primary and Alternate” tasks and limits the periodicity of TaskPairs (a base period multiplied with a power of 2).

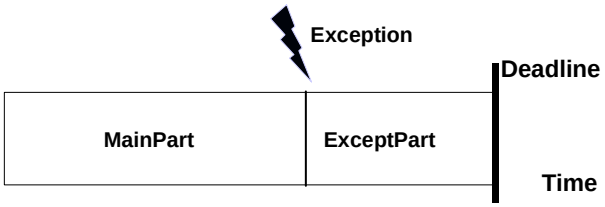


Figure 3.2: A TaskPair

This means, the fault-tolerance aspect of TAFT is provided by the EP of the respective TP. Whether more can be done than restoring a consistent and/or fail-safe state is very much application dependent. There are quite a lot of task types that are amenable to a fault-tolerant approach. For such tasks the EP is able to deliver a result that is still acceptable as output for the whole TP. To give an idea, a sketch of some typical examples of such types of application tasks are given:

1. Tasks having two versions: a primary and an alternative [Net96]. The result of both versions is acceptable. They differ, however, in that the primary version provides a better quality of service, whereas the timely execution of the other one can be guaranteed. In this case the MP would represent a best effort approach to maximize the resulting quality of service. It is aborted when it is time to allot the respective resource to the EP in order to ensure that the deadline is met.

2. Iterative tasks, producing an output the quality of which is the higher the longer they run. Many tasks having this property can be stopped early and

still provide useful output. Their quality is usually a monotonically non-decreasing function of the execution time. It is the job of the EP to evaluate the outcome of the corresponding MP and to initiate the appropriate actions to be done.

3. Best-effort tasks where, as the name implies, meeting of respective deadlines is not essential to the application. However, they do deal with time-varying data meaning that if not terminated timely their value for the application decreases and eventually will become useless. The goal in scheduling is to maximize the cumulative value, i.e. the sum of the values of the completed individual tasks.

The principal difference to conventional software fault tolerance measures is that in the TAFT approach exception handling is done as part of an entity, which is under the control of the real-time scheduler.

Time-Awareness

The MP contains the real application code and this is the code that is usually critical in terms of unknown or useless *WCETs*. At this point time-awareness becomes important. Execution times of MPs, by default, are interpreted as *ECET*s. The task-internal deadline of a MP is computed as the deadline of the corresponding TP minus the *WCET* of its associated EP.

In order to guarantee the timely execution of tasks, each newly arriving task has to undergo an acceptance test. Its outcome is positive, if the *ECET* of its MP plus the *WCET* of its EP can be reserved by the scheduler prior to the deadline of the TP. The approach taken by the adaptive TAFT scheduler to cope with the unreliability of a priori defined task execution times is to adjust its behavior dynamically in order to achieve a predictable overall behavior. More precisely, this means that the time reserved for the MainPart depends on the *ECET*s actually measured in the running system. It is still assumed that the $WCET_{EP}$ of an ExceptionPart is known, but this is not a severe restriction, as it is assumed to be very short. Thus, even a bad and very pessimistic execution time estimation should not lead to resource requirements that are comparable with those of the ordinary task ($ECET_{MP} >> WCET_{EP}$). This approach is viable if $WCET_T > ECET_{MP} + WCET_{EP}$. This means, the *WCET* of *t* is greater than its expected execution plus the time for a possible emergency procedure.

The behavior of the scheduler as described so far ensures that deadlines of Task-Pairs are never violated even if there is only a partial knowledge about their timing. But what happens with TAFT scheduling in overload situations, when there exists no schedule that can execute all requested tasks in time? As in other scheduling algorithms here the notion of "importance" (or "value") of a task comes into play. The scheduler has to guarantee that the executions of tasks are canceled in the reverse order of their importance. In TAFT scheduling the probability *p* (i.e. *k/n*) in $ECET_{t,p}$ is a parameter that can be used to express the impor-

tance of a task. As p raises towards 1, $ECET_{T,p}$ converges towards $WCET_T$. In the case of overload, scheduled TaskPairs with a value of p close to 1 will still receive enough guaranteed resources to successfully complete the MP (i.e. to achieve computational progress), while those with a lower value of p will probably run into more exceptions, resulting in degraded (or even no) computational progress.

3.1.5 Related Work

The adaptive, measurement-based scheduling approach of TAFT is related to a number of different areas in the real-time research, namely analytic WCET-analysis and its limitations, techniques for measurement-based timing analysis, other approaches that use online feedback of monitoring data in real-time systems, adaptive object-oriented systems in general, as well as real-time scheduling techniques that explicitly address fault-tolerance mechanisms. These areas of related work are covered in the following subsections.

Analytic Approaches for WCET Analysis

The basic work on WCET-analysis has been published in [Kli86] and [Pus89]. Kligerman and Stoyenko present in [Kli86] a restricted language, Real-Time Euclid, which was designed to make schedulability analysis possible under a number of assumptions about the system and process behavior. In [Pus89] Puschner and Koza propose the *Maximum Execution Time* (MAXT) concept, i.e. an extension to standard programming languages, which introduces *bounded loops*, *exceptions*, *markers* and *scopes*. Bounded loops are introduced that must either have a time limit or a limitation on the number of iterations. The basic idea of both research groups was to determine runtime boundaries of a program at compile time by avoiding any recursions, function variables, or jumps. If these conditions are satisfied by the software, the WCET can be calculated for basic constructs by calculating the execution time for the corresponding underlying machine instructions. However, this approach is limited by it ability to establish tight bounds on the execution time of the basic constructs. Exactly this has become hard on modern RISC and also CISC processors. Most of the research that has been published on WCET analysis for modern processors is focused on just one architectural feature: program path analysis, instruction caching, data caching, or pipelining. Combined analysis approaches tend to have either a high computational complexity or a weakness in some parts of the analysis.

In [Arn94] a technique is described to statically predict which instructions will be in the instruction cache during program execution. In this approach, called Static Cache Simulation, instructions are classified as *always-hit*, *always-miss*, *first-miss* and *first-hit*, by analyzing the control flow of the program.

For the analysis of data caches in [Bas94] a graph coloring approach similar to that used for register allocation in compiler construction is suggested. That ap-

proach tries to group variables based on temporal locality, i. e. variables that are accessed within one basic block are clustered in memory so that they fit into one cache block (spatial locality). Its main goal is to get more confidence on the estimated number of data cache misses.

[Lim94] presents an approach that addresses two aspects: pipelining and instruction caching. In that approach, a program statement is associated not only with a WCET, but with an abstract description of the current status of the pipeline and the instruction cache: the *worst-case timing abstraction* (WCTA). A program path can be analyzed by concatenating and pruning the WCTAs of its basic blocks.

The prediction of pipeline performance in combination with cache prediction is discussed in [Nil95] and [Hea94]. In [Nil95] the pipeline behavior is simulated for a given code segment. It introduces the *pipeline simulator compiler*, which uses a description of a processor to generate a program that simulates the execution of code on this processor. The main shortcoming of this approach is that cache prediction is weak and it is estimated that it is unlikely that even in an optimal case, the cache analyzer can predict more than 50% of the actual cache hits for realistic workloads. In [Hea95] the Static Cache Simulation is combined with pipeline simulation. However, it does not take into account data caching and instruction level parallelism.

In [LiM95] the program path analysis is modeled by integer linear programming (ILP). The instruction cache performance is also integrated in this ILP-model. Although the program behavior can easily be modeled by ILP, the analysis is likely to become inefficient for larger applications, since solving an ILP takes exponential time.

Measurement-Based Approaches for Timing Analysis

The general idea to use monitoring in real-time systems has already been exploited by other researchers. Haban and Shin [Hab90] used it to generate estimates for off-line scheduling decisions. This early work has been done on processor architectures that allowed for establishing tight bounds on the WCET by measuring the execution time of the basic blocks of the program. For more advanced architectures, methods known from dynamic testing have been studied by several research groups. An approach that regards the real-time application as a black box and that uses genetic algorithms to produce a timing estimate is reported to produce good results [Pus98]. However, no guarantee regarding the safeness of the results can be given, i.e. the results are not necessarily the *WCET*.

Other work tries to combine analytic and experimental methods. In [Pet99] a method for measuring the execution time of programs is presented. By analyzing the control flow graph, a reduced control graph is generated, which limits the paths to be measured. Using this information the object code of the program is instrumented and then measured. By measuring all paths the reduced control

flow graph indicates, bounds on the *WCET* are established without getting too pessimistic estimations. Similar to this approach, in [Lin00] a method for low-level timing analysis based on measurements of execution times of programs executing on the actual target architecture is proposed. The basic idea of the method is to derive a system of linear equations from a limited number of timing measurements of an instrumented version of the considered program. The solution to these equations gives the *WCET* for program fragments, from which the *WCET* of the entire program can be derived. However, both approaches produce safe *WCET*s only within the limits of their restricted system models, e.g. if the execution times of paths are input data independent. This is not true for many processor architectures, e.g. in case of arithmetical division instructions. Furthermore, the second method is not applicable to systems with caches.

In [Mos97] Moser et al. present a method for computing execution times based on a calculus for probabilistic density functions. These (discrete) functions are either extracted from a system by measurements or based on assumptions. The goal is to model the timing behavior of a real-time system by combining the probabilistic density functions of parts of the system. Statements on the dependability of the system are then based on the probability of a timing fault and not on fixed upper bounds like WCETs. No attempts are made to make this analysis available to the scheduler of the system.

All these described techniques do not try to utilize any object-oriented structure of the program. I.e. they can report on the timing properties of tasks (basically a piece of code, e.g. a procedure) but not on single instances. Also, all these approaches are off-line techniques.

Online Feedback of Monitored Data

The next step towards a system that can use monitoring data online to guarantee timing behavior has been done by Jahanian, Mok et al.. Their work started with the RTL (Real Time Logic) [Jah86] language for the specification of real-time system. The semantics of RTL is based on the occurrence of events that result from the execution of a real-time system (like the start and the end of code blocks or the assignment of values to status-variables). Timing properties can be expressed as the relationship between events. They developed algorithms for checking safety assertions [Jah87] and partial event-traces [Jah90] against RTL-specifications. In [Cho91] they propose a runtime monitor for the online-verification of properties of real-time systems. In this monitor, the timing constraints are divided into embedded constraints that can be verified immediately at their occurrence and monitored constraints that need to be verified by a separate monitor process in the context of the current event trace. The real-time processes are instrumented to generate the events of interest and also to verify the embedded timing-constraints. The idea of this monitor is to detect possible violations of predefined safety constraints in the running system as soon as possible. On the detection of such a violation a signal is raised. This signal might be used to trig-

ger some kind of exception handling in the monitored application itself or it may simply inform a human supervisor about the detected condition. This monitoring system has been extended by Raju et al. [Raj92] to distributed real-time systems. Each node of the target system has a monitor that collects event information generated by the user processes and verifies both embedded and monitored timing constraints. The monitors have to communicate in order to verify timing constraints that can only be checked with event-information of more than one node.

In [Mar91a, Mar91b] Marzullo et al. propose a reactive system named Meta. With Meta, a distributed system can be instrumented with a sensor and actuator abstraction that exposes the state of the system for purposes of control. Then, a control program can be written in an object-oriented modeling language that interacts with the instrumented system using guarded commands. The focus of Meta is more on fault-tolerance in general, as it does not address especially timing issues.

Adaptive Object-Oriented Systems

In [Bih91] Bihari and Schwan presented a model of an adaptive real-time system (RESAS). RESAS is a complete development and runtime environment that includes an object-oriented programming model, a representation framework, and an adaptation control system. The adaptation control system includes a data management system that stores static (from the compiler) and dynamic data (from the monitor) about the application's objects. Adaptations can be performed by manipulation of object shadows in the data management system, which in turn triggers the adaptation enactment mechanism. This work on adaptive objects has been extended in the context of the CHAOS real-time operating system kernel. In [Ghe93] the notion of policies associated with objects that intercept object invocation to make runtime decisions on invocation and object implementation is introduced. These policies can accept and interpret runtime attributes. Attributes expose selected aspects of object and invocation implementations. RESAS and CHAOS are generic and do not provide a predefined adaptation strategy. Because of their age, they were closed environments for adaptive real-time programming with no interfacing to object-oriented standards like CORBA or DCOM.

Work on the problem of integrating object-oriented components into real-time systems has been done in the context of real-time extensions of CORBA and Java. The obvious approaches aiming to build full CORBA compliant real-time ORBs, like TAO [Sch97] and the Real-Time CORBA specification [OMG99b, Sch00], is only a partial solution to the problem. They rely on static scheduling and they assume a closed real-time environment. The resulting limitations are becoming increasingly evident. CORBA-based applications tend to access objects that are outside the real-time domain, like data-bases or internet-based services, and these are usually shared among a large group of users and applica-

tions. Also they are often long living (That is why they were designed as CORBA services). In order to interface with these objects, a more adaptive approach seems to be more adequate, as it doesn't rely on strict assumptions about the behavior of these objects.

In the ARTDOM project [Kru98] at MITRE Corporation and the University of Rhode Island, a real-time trading object service has been developed. The service, when coupled with a corresponding traditional CORBA service, provides an adaptive binding service within CORBA. Thus, CORBA clients are bound to CORBA servers that can best meet their real-time requirements. This represents a best effort strategy and does not provide any guarantee to the client. In case of failing to meet the real-time requirements, the client system has no means like our FT scheduler to cope with that. The determination of the appropriate server is based on calculating the future requests (load) of the servers in question. The underlying timing parameters like request arrival times and (worst case) execution times are assumed to be known a priori.

Working groups dealing with real-time extensions to the Java platform are discussing approaches similar to the real-time trader [Foo99]. They call it a negotiating component that is able to negotiate with the runtime systems about resource requirements. It remains unresolved so far, whether such a component should be a low-level, integral part of the platform or only a "third-party API". The resulting real-time properties will depend very much on the outcome of that discussion.

Fault-Tolerant Real-Time Scheduling

Liu et al. defined imprecise computation [Liu94], where each task has a required part and an optional part. The optional part refines the computation performed in the required part, reducing the computational error. A modified task scheduler was used to allocate extra CPU capacity for the optional parts in order to reduce the overall computational error. Tasks have 3 levels: running, running with more computational error, and not running. Applications do not miss deadlines, and there is no deadline miss detection or notification. The major difference between the mandatory part in imprecise computations and the EP in a TaskPair is that an EP has to be executed if and only if its MP cannot be competed before the deadline of the TP, while the mandatory part in an imprecise computation is scheduled unconditionally before each optional part.

Lu et al. [LuS99] have developed a feedback driven version of EDF scheduling that uses a control theoretic approach to dynamically adjust the target CPU utilization based on direct measurements of the missed deadline ratio. Their work demonstrates the feasibility of this approach and shows that it works well under dynamic application loads. The feedback-driven EDF approach is limited to a single soft real-time policy wherein applications miss deadlines under situations of overload.

3.2 The Measurement-based Approach

The basic idea of this chapter is to extend the system support for time-awareness as presented in chapter 2, to an online feedback that allows for guaranteeing timing properties. With the ideas of soft tasks, *ECET*s, and TAFT scheduling as described in the previous subsection, the overall approach can be depicted as an adaptation loop as shown in Figure 3.3. At a high level, the complete real-time system consists basically of three interacting entities: the application, the monitor that observes the application, and the scheduler that controls it.

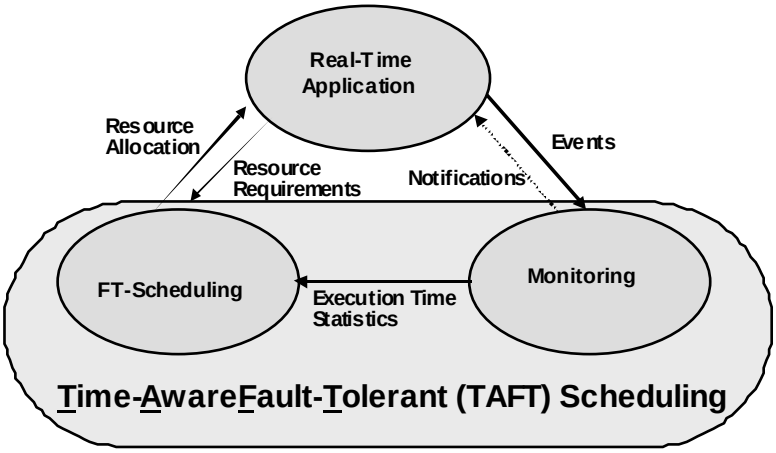


Figure 3.3: The adaptation loop.

The application is typically organized in objects, it is distributed, and it has to fulfill the specified timing constraints. In order to meet these constraints it has explicit or implicit resource requirements that are send to the system's scheduler. The monitor observes the actual behavior of the application. In order to do this, it collects events from the executing application. This monitoring is continuous, online, and also distributed. Internally, the monitor maintains a system model of the application. With the incoming events it updates state-information about the application entities (e.g. threads and objects). It provides online information about the observed timing properties to both, the application (for application-level adaptation) and the scheduler (for system-level adaptations).

The application receives (possibly asynchronous) notifications from the monitor as soon as predefined timing conditions are detected to be fulfilled. This kind of time-awareness allows for direct adaptations of the application. It can be viewed as a part of the application logic that has been delegated to the system's infra-structure as the infrastructure can do the job more efficiently, with less intrusion, and in a generic way for a huge class of application.

The scheduler uses the input about the timing of the application from the monitor as parameters for optimizing its resource allocation decisions. As depicted in Figure 3.3 the monitor and the scheduler together can implement TAFT scheduling: the monitor provides an FT-Scheduler with the required execution time statistics of the *ECET*s and the scheduler dynamically adapts the guaranteed execution time of the MPs according to their current behavior, their importance, and the current overall load situation.

The following subsections will present the details of the measurement-based approach. Focus is put again on the monitoring component and especially on applying the concepts to object-oriented systems. Firstly, as already sketched above, ECET-analysis as source of online timing approximations for the TAFT scheduler will be described in more detail. Then, an advanced model for the analysis of the expected termination time will be introduced that allows for saving resources by a notification of "hopelessly late" MPs.

3.2.1 ECET Analysis in Object-oriented Systems

Up to now, object-orientation, despite its doubtlessly existing advantages for software engineering, has been regarded mainly as the source of problems in real-time computing, not as the solution. Due to its feature to hide implementation (and thus timing) details and its ability to integrate third-party components, it has aggravated the problem of application code with unknown or completely unrealistic *WCET*s. But now, object-orientation can also help to solve these problems, at least partially. One of its main features, namely the ability to capture much of a program's data-dependencies in syntactic categories, will be exploited to provide better timing estimates that it would not be possible in a purely procedural structure.

System Model

Consider the following system model: the distributed application is modeled as a 10-tuple $(C, M, f_m, T, O, N, f_c, f_n, A, f_a)$ with

C is a set of classes,
M is a set of methods,
f_m: $C \rightarrow 2^M$,
T is a discrete time-base (for simplicity the natural numbers),
O is a set of objects,
N is a set on computing nodes,
f_c: $O \rightarrow C$,
f_n: $O \times T \rightarrow N \cup \varnothing$,
A is a set of activities, and
f_a: $A \times T \rightarrow \{O \times M\} \cup \{\}$.

An application consists of a set of classes *C* with a set of methods *M* and a set of objects *O*. The function $f_m(c)$: $C \rightarrow 2^M$ maps a class $c \in C$ to the subset of methods that are members of *c*. This describes the static structure of the application. More detailed relationships (inheritance, use-relation, etc.) as required for object-oriented design are not considered here and they are not needed for the following considerations.

The dynamic structure of the system is described by the remaining components: the function $f_c(o)$: $O \rightarrow C$ maps an object $o \in O$ to a class $c \in C$, i.e. it determines which object belongs to which class. Consequently, $f_m(f_c(o))$ denotes the methods applicable to object *o*. The function $f_n(o,t)$: $O \times T \rightarrow N \cup \varnothing$ describes the location of object *o* at time *t*. If the object has not yet been created at time *t* or already destroyed, f_n maps to the empty set. Note, that the notation of function f_n allows for the migration of objects, i.e. the mapping of an object to a node may change over time. The described monitoring system does not yet support this. Instead, whenever an object is first seen at a node, it is treated as a new object. However, this is not a severe limitation, as most systems do not support migration and even if, the timing of a migrated object will probably change, thus new data on its behavior is required anyway.

Finally, activities in *A* are described by $f_a(a,t)$: $A \times T \rightarrow \{O \times M\} \cup \{\}$ that maps an activity *a* at time *t* to a certain method in a certain object. If the activity is not yet started or already terminated, f_a maps to the empty set. Activities are "distributed threads" as discussed above and they may be periodic or aperiodic. Activities are considered to behave like a usual execution with a call-stack, i.e. an activity *a* starts at some time $t_{0,a}$ in a top-level method-execution $(o_{0,a}, m_{0,a})$ and for each subsequent clock tick it either stays in this method or switches into another methods following the syntactic rules of nesting and concatenation as depicted in Figure 3.4.

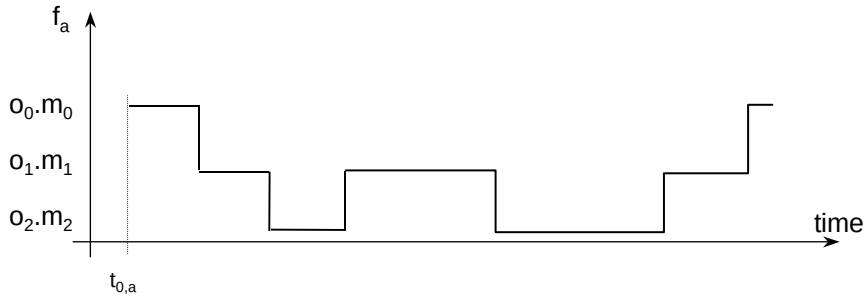


Figure 3.4: The stack-like object invocation sequence of an activity

Each change in function f_a from one point in time to the subsequent denotes either the invocation of another method on behalf of the current method or the return from an invoked method to its caller. "Forking" of activities, i.e. the paral-

lel execution of multiple threads of control is not considered. Instead, a new activity starting in the current object is created if one activity initiates a new thread of control.

Methods – Entities of Monitoring and Scheduling

The changes of the value of function f_a correspond to the events generated by the different instrumentation techniques for object-oriented systems as described in chapter 2. Whenever a thread (on behalf of an activity) enters or leaves an object's method, an event is generated. This means, a monitor is able to determine the timing of every method-invocation and, instead of just presenting this data in a graphical presentation as discussed before, it can also process this data in order to provide statistics on their timing.

Also, methods-executions are the natural entities of code for scheduling. Ultimately, the scheduler has to schedule periodic or aperiodic activities, corresponding to the classical notion of tasks in non-object-oriented systems. However, the code of activities is structured by methods. An activity starts and ends in a top-level method and in between it is organized by the function f_a into a concatenated and nested sequence of methods. In all considered object-oriented systems, method-invocations also denote the boundaries of objects and thus possible node boundaries. If the timing of certain parts of the application is unknown due to unpredictable network-delays, this is manifested in the timing of those methods that are called remotely. Also, whenever a heterogeneous object-oriented system has to invoke a third-party service with unknown timing, this happens at method boundaries. Therefore, methods are the natural entities that have timing attributes like execution times or other resource requirements.

The state of objects is accessed and changed by invocations of their methods. If an object is implemented to provide soft task behavior, this is best encapsulated in the code of a method. Whatever happens exactly inside a method is invisible to the caller by the rules of implementation hiding, but it is guaranteed that the object is in a consistent state after the method invocation has terminated in time. Therefore, methods are the ideal entities to be organized as TaskPairs.

From the view of an object that invokes another object's method with unknown timing-behavior, it is also desirable to encapsulate this invocation in a TaskPair. Like with a classical timeout it might want to limit the maximum time spend in the invocation. However, with a surrounding TaskPair it can also specify the emergency action that is executed in case of the failure of the invocation and it can put a guaranteed upper bound on the complete transaction. Again, knowledge about the expected timing of the actual invocation will help the scheduler a lot to ensure computational progress.

Putting this together, an implementation of the adaptation loop as depicted in Figure 3.3 for object-oriented systems requires a fault-tolerant scheduling of methods and a monitor that provides timing information on a per-method basis.

Objects – Capturing Data-Dependencies

Up to now, the main focus of interest has been on methods. The method's code belongs to the static structure of a system and one could expect that timing attributes of methods are static as well. However, in general, the timing of a piece of code does not only depend on the statements to be executed, but also on the data that the code manipulates. This is obvious if the code contains conditional branches and loops, but also the timing of straight-line code can be affected by data-dependencies (e.g. in variable timing of arithmetic instructions or cache issues). For tight WCET-analysis and also for timing estimations it is important to to minimize the influence of data-dependencies. Therefore, [Pus89] already proposed meta-statements that exclude certain input-data for timing analysis. But as long as procedures can operate on an unknown set of input-parameters, it remains hard to take data-dependencies into account in a static analysis. However, in contrast to procedures that are pure static code, methods do explicitly belong to objects.

An object-oriented environment possesses the desirable property that data dependencies in execution time behavior are partly tied to object instances. This is true for all data-items that are part of the object's state. It is not true for the remaining input-parameters. This means, the ability to tie data-dependent timing behavior to objects varies with the design of the object-oriented system. If the main paradigm of the system is data shipping, it will be less promising than if it were function shipping. However, in a distributed environment, where communication costs are still a dominate factor, object state tends to be bigger and function-shipping can be expected to be the first choice.

Since an object is an entity of data and code, variances caused by data-dependencies can be captured by monitoring and analyzing the timing characteristics of each object separately. Objects of the same class, e.g. a class "List", may expose completely different timing behavior depending on their internal status (e.g. list length) and the environment (e.g. communication costs). Thus, monitoring at the class level only, as would be the possible approach in a non-object-oriented environment, would encounter variances that automatically disappear when object specific monitoring is applied. Thus, object-orientation helps to achieve more accurate timing estimates for methods as it allows to monitor them also on a per-object basis.

3.2.2 Maintaining Timing Statistics for ECET Analysis

In order to provide the FT-scheduler of the TAFT system with reasonable timing estimations, for each pair (*o*, *m*) the $ECET_{(o,m),k,n}$ has to be available, i.e. the minimal execution time that was needed to successfully complete at least *k* out of the last *n* executions of method *m* in the context of *o* (independent of the activity that actually executed *m*). From continuous monitoring, the start- and the end-events of the method (*o*, *m*) are available and by simply computing the difference

between their time-stamps, the overall execution time (including all blocking times) can be computed easily. Blocking times can be eliminated by additionally taking thread-switch events into account as described below.

In the simplest case one could assume executions times as being normally distributed and compute the ECET from the constantly updated average and variance for a desired execution completion rate *k*/*n*. While this approach requires only a minimum of additional state per pair (*o*, *m*), it will fail for most realistic execution-time distributions. As stated in [Mos97] also other well-analyzed distributions, like e.g. negative exponential distribution, tend to be a bad approximation for the real behavior of code.

Therefore, a more realistic approach is to maintain an efficient discrete representation of the measured distribution, as depicted in Figure 3.5. It holds the execution time density for the *n* most recent executions of a method *m* of object *o*. The maximum seen execution time $max_t(o, m)$ is divided into *l* equally sized slots. A slot *i*, with $0 \le i < l$ represents the number of executions that had a duration in the time interval $]i * max_t(o, m)/l, (i+1) * max_t(o, m)/l]$. Whenever a new monitored execution time comes in, it is added to the slot that represents its execution time. In turn, the value that had been added *n* events ago is removed from the representation.

The $ECET_{(o,m),k,n}$ is then determined as the time, which *k* of those execution times lay within. It is computed by summing up the number of executions starting from slot 0 upwards until *k* is reached and taking the upper time limit of this slot:

$$ECET_{(o,m),k,n} \le (i+1) * \frac{\max_t(o,m)}{l} \text{ with } \sum_{x=0}^{i} slot(x) \ge k$$

This is basically again the integration in order to get from the density function to the distribution. The density has been chosen as primary representation as it allows for a more efficient online update when events with new timing data have to be integrated As the number of slots *l* is constant, the computation of ECETs can still be done in O(1) time.

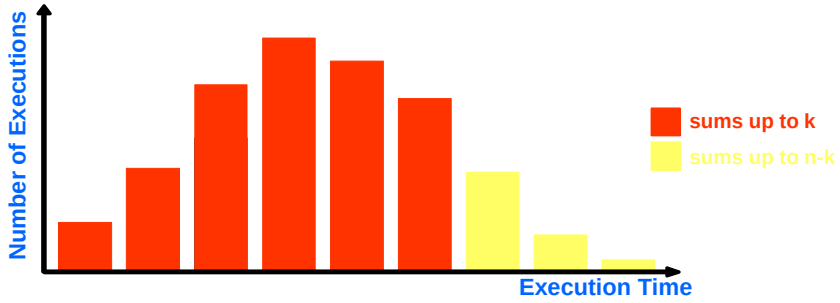


Figure 3.5: ECET evaluation with a discrete representation of the distribution

While this kind of statistics is easy to maintain and to evaluate, it has a major drawback with respect to space complexity. By construction the size of the data-structure is in O(|*O*| |M|). This means the size of the data scales linear with the number of (monitored) objects, i.e. the size of the application. If however, it also has to maintain an event-history (basically a queue) of length *n* per object and method (in order to remove the outdated events) the scaling factor becomes rather large. Figure 3.6 shows an example where a new event with execution time of 4 comes in and is queued, while the oldest event with execution time 6 is removed from the history and subtracted from the density representation. A history of length 100 or even 1000 will probably be no exception, which means that monitoring data might have a bigger memory footprint than the monitored application itself, a clearly undesirable effect.

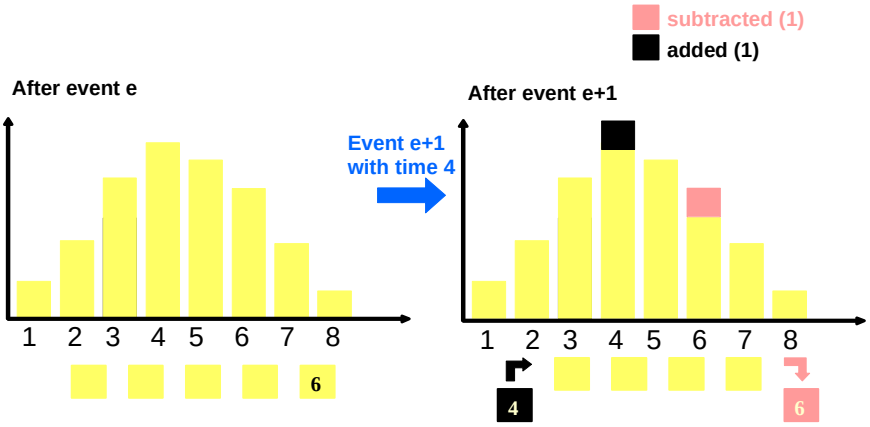


Figure 3.6: The event-history representation of the n most recent execution times

Therefore, currently a negative exponential fade-out algorithm is used to reduce the effect of old events. It approximates the density of the last *n* executions by subtracting one n^{th} of each slot's value before the value of a new event is added. This equals a multiplication with (*n*-1)/*n*. Figure 3.7 depicts the same example as in Figure 3.6, but this time with the exponential fade-out algorithm. The state of the required data structure consists only of the slot values, which is more acceptable than the full history as it is in most cases at least smaller than the observed object itself. While the negative exponential character of this algorithm never leads to a zero slot-value, there is a threshold. If the value of a slot drops below this threshold, it is set to zero. This is important in order to allow for an adaptive re-scaling of the slot-sizes as described below.

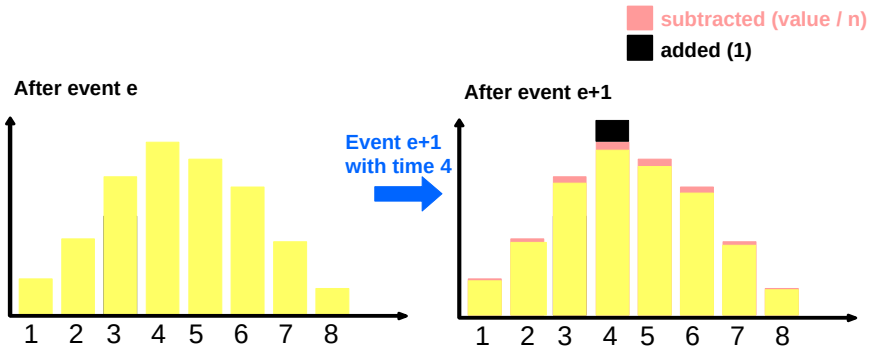


Figure 3.7: The negative exponential fade-out algorithm.

If for a method (*o*,*m*) no events have been collected so far, the $ECET_{(o,m),k,n}$ cannot be computed using the algorithm described above. In order to initialize new methods with reasonable timing defaults, an additional record on ($f_c(o)$, *m*) is maintained. It holds the execution time density of *m* in the context of the class *c* of *o*. It is updated each time an event of method (*o'*,*m*) with $f_c(o') = c$ is observed. This per-class statistics refers to the static part of the application's structure, while the per-object statistics reflects the behavior of the dynamic entities. However, both statistics are updated online and may change over time.

As the overall maximum execution time of a method (*o*, *m*), the $WCET_{(o,m)}$, is still unknown, it might happen that an event comes in reporting on a new execution time that larger *t'* than the current $max_t(o, m)$. In this case the density representation has to be re-scaled. This can happen either by setting $max_t(o, m)$ to *t'* or by simply doubling the current $max_t(o, m)$ until it is larger than *t'* (see Figure 3.8). In both cases the slot sizes have to be adapted accordingly and the current values of the slots have to be distributed proportionally among the new slots covering the same ranges. The first solution has the advantage that it provides a maximum resolution with the fixed number of slots (its slots cover only the range of actually seen execution times). The latter has the advantage that it causes probably less future re-scaling actions, as it doubles the range each time.

Re-scaling in the other direction becomes necessary if the density function shifts to the left, i.e. the monitored execution times are becoming shorter.

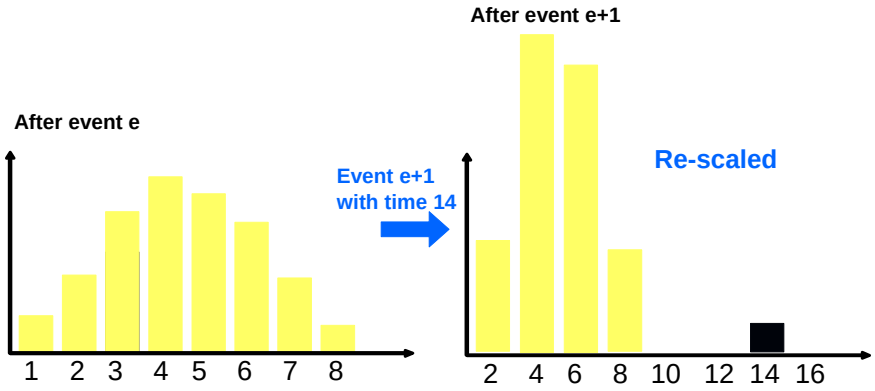


Figure 3.8: Adaptive re-scaling of the density representation

*ECET*s and the presented data structures so far refer to "successful" completions of a method (*o*, *m*). When scheduled with the FT-Scheduler this means a normal termination of the MP. Sometimes reasonable new *ECET* values cannot be provided because all of the recent executions of the MP of (*o*, *m*) have been interrupted by EPs. This typically happens, when the last computed ECET value is for some reason suddenly by far to small for the next required computation of the same (*o*, *m*). In this case, another heuristic is required. The proposed approach is to exponentially increase the formerly computed *ECET* (by a factor of 2) when the *k* most recent executions have failed in order to find quickly a feasible upper bound that can be refined by subsequent monitoring. The same heuristic is applied initially on the first occurrence of an event reporting on a new method, when there is even no default value from the per-class statistics available to initialize the timing data.

Just to clarify, it should be recalled that despite all approximations and heuristics, the job of the *ECET* computation is to provide reasonable estimations in order to allow for an effective scheduling and to enable computational progress in the long run. Safety and timeliness are still guaranteed by the scheduler and do not rely on the value of the *ECET*s.

Considering Blocking Times

Accounting the timing of methods simply from the execution of the first statement until the termination leads to a measurement that includes all blocking times, regardless whether this blocking is caused by preemption or by synchronization conditions. As a result, the calculated timing predictions would also report on this overall execution time. For the purpose of predicting the timing of remote

services (that are not under the control of the system's scheduler) this is the appropriate measure: the real-time system is not really interested in the internal reasons for the observed timing. It needs the overall time until completion for choosing proper timeout values.

Things are different if timing prediction for methods of local objects is considered. Here, the scheduler is interested in the pure CPU-time (as the definition of *ECET*s requires) as it is used as planning base for assigning CPU-resources to the executing activity. Still, the overall execution time including the blocking time is helpful as it provides a hint whether a method has a chance to meet its deadline even facing non-trivial synchronization conditions (see also the next section). Resulting from this, the *Expected Case Runtime* (ECRT) can be defined as:

$ECRT_{t,p}$:= The overall time (including all blocking time) that is needed by task instance *t* in order get a probability of *p* that *t* is completed.

From the view of statistic evaluation the same mechanisms and data-structure as described above for the *ECET*s can be applied to *ECRT*s in order to provide predictions based on the recent behavior of the task. Also, the monitoring systems from chapter 2 can support both, *ECET*s and *ECRT*s. The overall timing information required for *ECRT*s is provided by the middleware- and language-level instrumentation. It reports on the start and the end of method executions. For determining the pure CPU-time, the instrumentation at operating-system-level is required. Using the information from the events reporting on thread-switches and interrupts, the actual CPU-time spend on a single thread can be calculated easily. MagicZoom as described in subsection 2.5 records and visualizes exactly this information.

For the purpose of ECET-analysis the sensor-code can be simplified: instead of recording each thread-switch in the event-trace and reporting it to the monitoring console, it can account for the CPU-usage of a thread directly at kernel level, like e.g. the g*etrusage()* system-call does for Unix processes. A virtual clock per thread is maintained and this clock is used for an additional time-stamp of the method-start and method-end events. Again, it turns out that for a successful monitoring of real-time systems an observation at all architectural levels is required.

3.2.3 Early Detection of Timing Faults

Up to now, *ECET*s have been considered as input for the scheduler before it generates the schedule. Given that the prediction of *ECET*s works well, i.e. the approximation of the probabilistic value by the statistics over the recent *n* executions leads to correct results, this enables the system to create efficient and resource-saving schedules. However, by the definition of $ECET_{(o,\ m),p}$, there is the probability of 1-*p* that the execution takes longer than $ECET_{(o,\ m),p}$ suggests. In all

these cases even a guaranteed execution time for the MP can lead to an abort of the MP and an exception handling by the EP. Although this is not a safety problem, due to the guaranteed timely execution of the ET, it would be a waste of resources. The CPU-time spend on the MT is lost and has not lead to any computational progress. While this is the price to pay for the use of estimations instead of hard bounds, there might be the chance of avoiding unnecessary waste of cycles.

This is especially an issue when the considered methods themselves conditionally invoke further objects. Consider a case where an object's method exhibits a timing distribution as depicted in Figure 3.9. Depending on the input parameters or the internal state it sometimes invokes an external object, in most cases it does not. This behavior leads to a timing distribution with at least two peaks that may be separated by a large time interval (consider the difference of a local computation and an external call, e.g. to update a database). If now the *ECET* of the top-level method is requested by the scheduler, the result might be (as in Figure 3.9) that second (small) peak is completely cut off. This happens because only a very small percentage of executions exhibits this extreme timing behavior.

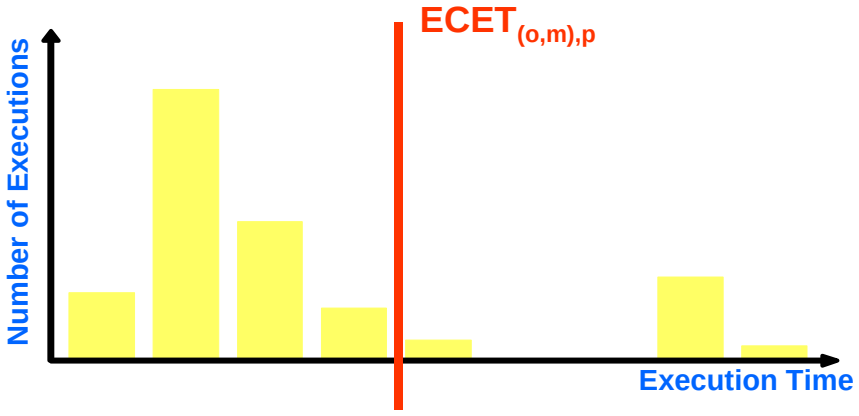


Figure 3.9: A Timing density with high variance caused by data-dependent branching

When looking at the code of the method, it might be easy to identify the code that is responsible for this behavior. Typically, it is data-dependant branching. Moreover the branch, which takes significantly longer, often invokes some other methods (possibly on remote nodes). How can this behavior be captured by ECET monitoring?

The idea is to re-calculate estimates of the remaining execution time based on the state reached so far. If the calculated remaining execution time exceeds the reminder of the originally calculated ECET, the current execution is probably an

instance where the ECET was too small. Since this can now be detected before the actual end of the original ECET is reached, possible EP-aborts are detected early. As soon as the system detects that the activity will probably violate the overall deadline, the calling object is informed and, depending on the new estimate, it may decide whether to abort the call or still to wait for its results.

In order to discuss this simple early detection mechanism in more detail, the concept of the *Expected Case Termination Time (ECTT)* is introduced. The *ECTT* uses the concept of *ECET*s to predict when an execution will probably terminate. As depicted in Figure 3.10, the *ECTT* of a method (*o*, *m*) is defined to be $ECET_{(o,\ m),p} + t_{0,(o,m)}$, i.e. the starting time of a methods plus its ECET. This results in the absolute time when method (*o*, *m*) is expected to terminate.

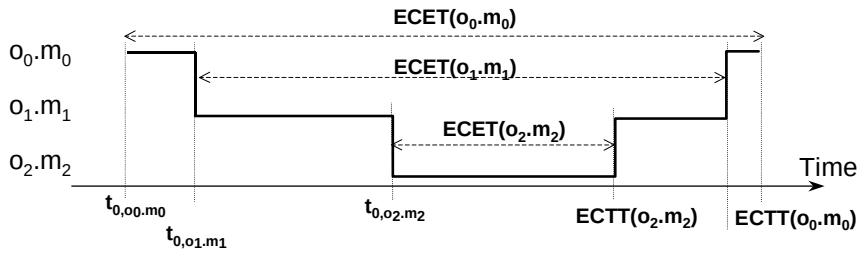


Figure 3.10: The concept of the ECTT

However, the ECET of a method changes slowly and during the execution it is constant. Therefore, the notion of an *ECTT* alone does not reveal any new insights that can help in detecting extreme timing behavior of a method. Here again, object-orientation can be of great help: assume only the timing distribution of individual methods is available, regardless of the method internals, e.g. nested method calls. By this, each nesting level has its own *ECTT*, i.e. each currently invoked method in an activity can be viewed independently. Each time a new nesting level is reached, i.e. a method is invoked, the monitor can concurrently check the compliance of the new *ECTT* of this nesting level with the guaranteed execution time of the complete activity. This can be done in two different ways.

In the simplest solution, the $ECTT_{(o,\ m),p}$ of each nesting level is computed at the time of the entry in this method and checked against the guaranteed execution time of the activity (or directly the deadline, if *ECRT*s are considered). If it is greater, the ECET prediction at least at this level has serious doubts that the ECET for level zero was enough. This kind of probable timing fault detection can capture at least the case discussed above, where a data-dependency leads into a branch with a longer-lasting remote invocation. As shown in Figure 3.11 for method $o_2.m_2$, the monitor will detect the contradiction in ECTTs at level 0 and 2 as soon as it gets aware of the invocation of $o_2.m_2$. The current ECTT for $o_2.m_2$ is later than that of o_0,m_0. While the first fits into the deadline, the second does not.

Now, the activity might be informed about the possible problem of wasting resources. One major drawback of the described technique is that it only detects problems that become evident in the execution time of one method (the one that finally exceeds the available time). At most the duration of this one method can be saved, when a MP is aborted before it has finally exceeded its quantum. This does not help a lot when several consecutive, comparably small methods exceed their predicted *ECET*s.

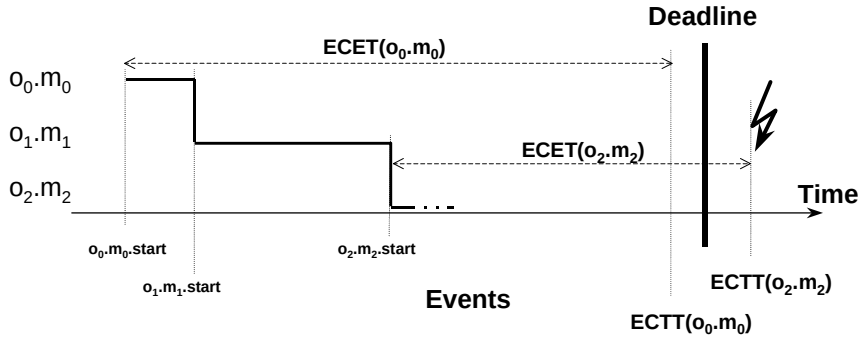


Figure 3.11: Detection of probable timing faults using ECTTs

An extension to the method sketched above that addresses this case is shown in Figure 3.12. The timing difference (ΔT) between the *ECTT* and actual termination time for each nesting level is computed. The detection algorithm sums up these deltas along the execution of one nesting level, i.e. the deltas of subsequent invocations in one nesting level are added. This means, the time gained and extra time spent in methods are summed up. Each time a method terminates, the result is added to the *ECTT* of all methods in the current nesting hierarchy. If one exceeds the overall execution time, a violation is predicted. In Figure 3.12 at the end of $o_2.m_2$ one might want to raise an exception as the top-level *ECET* of ($o_0.m_0$) plus the extra time consumed by $o_2.m_2$ is greater the overall deadline.

The problem with these two models is that they assume that method executions are independent from each other. In particular, they do not account for the fact that different execution paths might or might not be reachable from the current state. It is very difficult to calculate the reliability of the decisions resulting from the application of the used heuristics. It can be concluded the existence of multiple alternative completion paths for one activity at any point in the control flow graph is counterproductive for the simple model. It has to be emphasized that objects with statistically independent execution times of all method segments are quickly analyzed using this model. An approach that models the different execution paths in more detail using a Markov-model is proposed in [Ger97a]. However, this approach relies on a considerably larger monitor state for each observed method and it requires the observation of additional events reporting on branch decisions. While a tool as mc4p can produce these events, the computa-

tional effort in the monitor is not justified by the CPU-cycles that can be gained by early detection of possible timing errors.

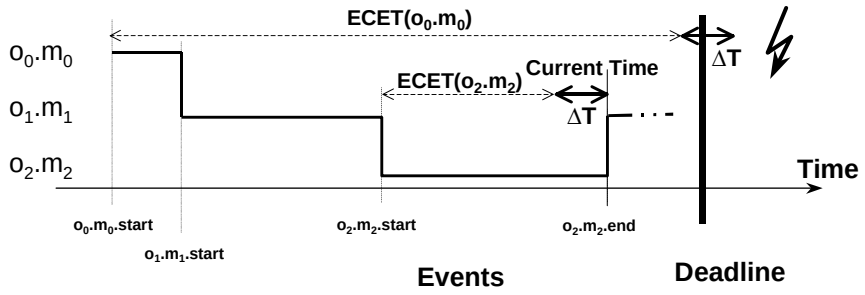


Figure 3.12: Detection of a probable timing fault with time-deltas

3.2.4 Adapting Granularity

Despite all considerations that try to minimize the space and time complexity of the ECET prediction, it is still a considerably amount of processing power and memory required to implement the algorithms presented in this subsection. Depending on the size of the monitored object, the amount of resources spend on the monitoring representation might become bigger than the requirements of the original object. This is a clearly undesirable state that has to be avoided.

A key in achieving a balance between the supporting monitoring system and the system under test is the use of the right granularity of the monitoring. The granularity can be adjusted with parameters at various levels.

1. Class level: As discussed in subsection 2.3.1 not necessarily all classes and thus all objects have to contain the same sensor code. Small-size objects, like e.g. arithmetic types, or all kinds of private helper-objects that do not reveal their interfaces to external classes, are good candidates for being skipped in the instrumentation process. Their execution time will than be added to the objects that use these classes. Typically, scheduling support is only required for objects that are about as big (in terms of execution time) as schedulable entities. For current systems this means in the order of milliseconds to hundreds of microseconds.

2. Object level: Even if the code of a class contains the instrumentation code, not all objects of this class might be of interest. If only a small number of the objects is involved in time-critical activities, it reduces the amount for required monitoring resources, if only these objects contribute to the load on the monitoring system. Objects might also be out of the scope of the time-awareness component if they belong to hard or essential tasks. In this case

additional information on their timing might be interesting, but it cannot contribute to an improvement of the systems runtime behavior. Their schedule is determined by static *WCET*s.

3. Method level: The same that applies to classes is true for methods as well: not all of them are necessarily interesting for being monitored. Only the external and scheduling-relevant methods need timing estimations. The others can be accounted for the invoking methods. Sometimes it might be also an option not to distinguish between the different methods of an object. If e.g. an object is actually a wrapper for a remote invocation of a small (in terms of execution time) object on another node, it is not the actual execution time of a single method that the scheduler is interested in, but the overall time of the RPC that is dominated by the communication overhead. In this case the *ECRT* (that includes waiting for the return of the RPC) of all methods of this object can be accumulated and analyzed in one density function.

4. Analysis level: Depending on the required accuracy and perhaps on the quality of previous ECET predictions, the type and the granularity of the statistics used for analysis can differ. For a first approximation a simple normal distribution with mean and variance may be enough. Only if it turns out that the variance becomes to high, a switch to a discrete density representation is initiated. Within the discrete density representation the number of slots is an important factor that scales resource consumption versus accuracy.

While the first mainly influence the amount of generated events, the last has impact on the event-processing component. The number of events is controlled by static and dynamic filters applied during instrumentation and event generation. The required mechanisms have been described in chapter 2. The design of the components that allow for an effective and flexible processing and analysis of these events is described in the following subsection.

3.3 The Implementation Architecture

This subsection describes a system-architecture that implements the measurement-based approach for real-time system and it describes the components and their tasks. The complete system is depicted in Figure 3.13. It is divided into the user-provided real-time application and the runtime system.

The application consists of the user-written code implementing the application objects plus the complete environment that is required to execute these objects. The application has to fulfill the end-to-end real-time requirements. Considering e.g. an embedded real-time application that has to interact with external CORBA services, the application consists of the user-written code of the embedded application, the CORBA objects implementation, (preferably coded in an object-oriented language) the CORBA-stubs (generated from the IDL), the object-adapter, the ORB itself, and the operating system that hosts these components.

This complete system is augmented with the sensors of the monitoring system. These sensors report on events from all architectural levels and forward them to the local runtime system.

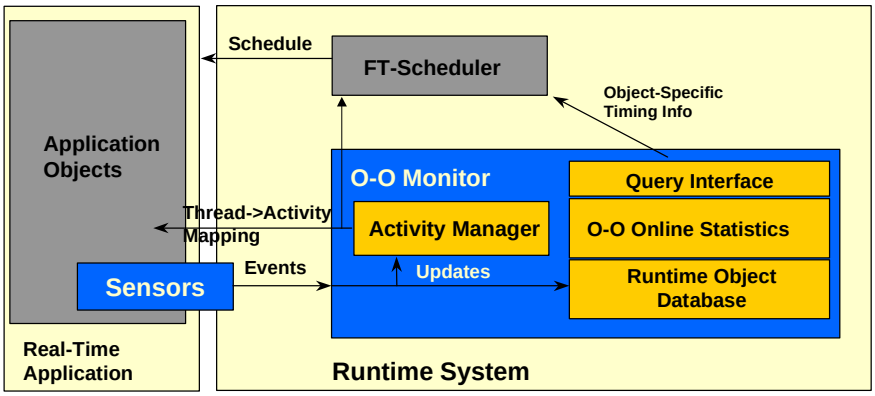


Figure 3.13: The components of the implementation architecture

The runtime system contains all components that are required beyond the standard real-time operating system functionality to close the feedback loop of the adaptive system and to implement the TAFT-Scheduler. It consists of the monitor and the scheduler. The internals of the scheduler are beyond the scope of this thesis and are explained in [Str95, Net97c]. The monitor uses the same basic mechanisms for instrumentation, for detecting, time-stamping, storing, and retrieving events as the monitors described in chapter 2. However, each local instance now contains additional components.

- The Runtime Object Database (*RODB*) manages the storage of the required per-object status.
- The Online Statistics component performs the computations described in the previous subsections for determining the *ECET*s of methods.
- The Query Interface provides an object-oriented and properly synchronized access for local and remote schedulers to the Online Statistics.
- The Activity Manager traces activities and provides this abstraction for those application environments that do not support it by default.

The architecture is generic in the sense that it does not rely on a specific object implementation for the real-time application side. The interaction between these two blocks happens only via events (that report about the application's timing) in one direction and via resource assignment and notifications (that actually influence the applications behavior) in the other direction. Principally, any object-oriented application can be made adaptive by this architecture as long as is can be augmented with the required event-generating sensors and provides computa-

tional entities that can be scheduled by the TAFT-Scheduler. In the following subsections, the distinct components of the monitor will be discussed in more detail.

3.3.1 Runtime Object Database

The Runtime Object Database (RODB) is responsible for organizing and storing the information gathered about the objects of the real-time application. It interfaces to the remaining components of the architecture through the event buffer and the Query Interface. The RODB server-processes also execute the code of the Online Statistics components. While the RODB is an object-oriented database, it was a design decision, not to implement it using a generic OODB system. This is mainly because of two reasons:

1. The main features offered by an OODB system are not in the focus of interest in this implementation: neither persistence nor concurrency management are required in the first place. Thus, any performance or memory penalty imposed by an OODB system is clearly unnecessary overhead.

2. The structure of the RODB is quite regular and the search pattern is always the same. Therefore, there is no indication, that an OODB system can achieve a better performance for the most critical activity, namely event processing.

The RODB is currently implemented in C++ and can be run on Sun Solaris, Linux, and, by encapsulation of the thread and shared memory APIs, also on Windows NT.

The RODB is distributed in a sense that each node holds the current monitoring data obtained from the application objects it is hosting. The main data-flow is generated by the events and the queries of the local scheduler. Therefore, any other partitioning of functionality or data would tremendously increase the amount of data to be shipped over the network. This would be a clearly undesirable effect, as communication is still a bottleneck in a distributed system.

In order to maintain the per-object timing information, the RODB uses the events from the real-time application to keep track of the dynamically evolving object-space. From the static program analysis the RODB knows about the class-structure. Static program analysis for language-level objects is done by the (pre-) compiler. The required information which classes generate which events are available via the mc4p name-server. For middleware-objects the IDL-compiler typically extracts the static class information and stores it in a database that is accessible for application processes. In CORBA this is the Class Repository and DCOM puts this information in the system’s “Registry” database. With this knowledge, the RODB interprets the event-stream and constructs a model of the current object-space of the application. In a way the RODB mirrors the knowledge that is also held in the application's internal data-structures.

Data-Structures

According to the static and dynamic structure of object-oriented programs the RODB is organized in two separate parts (see Figure 3.14). The dynamic part is supposed to reflect the application *objects*. The second part accumulates all run-time-data according to the objects' *class* membership. It is called the static part. This part does not only recollect the currently running application's object behavior from the classes-view, but it is also supposed to accumulate the classes behavior over multiple runs of the target application. The class-part of the RODB can be made persistent by writing it to a file before shutting down the service. This makes sense, as class-behavior is usually not dependent on the current instance of the system but reflects a general property of the code. As object-oriented code usually changes only slowly over time, the static part of the RODB can even persist minor code changes due to bug fixes or further development. It is the decision of the user when to reset the stored class information in the RODB.

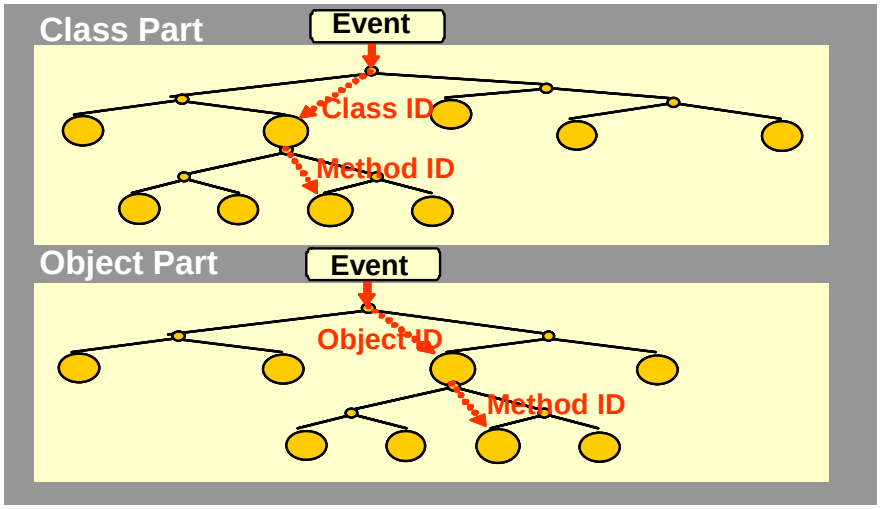


Figure 3.14: Data-Structures of the RODB

Both parts of the RODB are internally structured in a similar manner. They are optimized for a two level search (plus a root at level zero). The first level is indexing the objects (respectively classes) and the second is matching the methods. Both levels are internally organized as balanced search-trees (see Figure 3.14). Thus, each node at level two represents one method of the system, either regarded as a piece of code in the static part, or in the dynamic part as element of an activity when executing in an object. The node at the first level represents objects and classes of the monitored system. Each object node in the dynamic part also contains a link to the corresponding class node in the static part.

Each node at level zero (root), one, or two can hold *evaluators*. An evaluator is basically an event-consuming entity. More concrete, it is an object of any subclass of the abstract base-class *TstatEval* (see Figure 3.16). Each node can contain an arbitrary number of (different) evaluators. The types of evaluators to be maintained for a given node are determined by the father node at the next higher level. They are inherited at the moment of the node creation. In addition, evaluators can be added and removed dynamically.

Event Processing

Whenever a new event reporting on method (*o*,*m*) comes in, it is the job of the RODB to route it to the evaluators that need to know about it. These are the evaluators for object *o*, for method *m* of object *o*, for class $f_c(o)$, and for method *m* of class $f_c(o)$. The routing of events is done by searching in the balanced tree data-structures according to the class, object, and method identifiers. Each evaluator receives a copy of the event. While class- and object-level evaluators do not distinguish the different methods (as this is done as the next level), these evaluators might still be useful. They receive events of all methods of an object/a class. Whenever a search leads to an empty result, i.e. a class, an object, or a method has generated an event for the first time, a new node is created and initialized with cloned instances of the evaluators of the next higher level. Thus, the evaluator set of a higher-level node serves as template for the creation of new nodes. In the simplest case, the evaluator types of all nodes are equal to those of the root.

The complexity of this search process grows with O(log *n*), *n* being the number of different object-contexts, due to the tree-algorithm it uses (given that the number of classes and methods is constant in a running system).

Execution Model

The RODB is implemented in a separate user-level server process. It replaces the External Server component in the JewelNT and MagicZoom design. It takes over its functionality of extracting the events from the local event buffer. Figure 3.15 depicts the structure of the RODB server process. Two threads are executing in the server. All interfacing is done via shared memory interfaces in order to avoid additional synchronization operations and to minimize the overhead of interprocess communication.

The Update Thread is responsible for the actual event processing. It collects the events from the event queue in the share buffer, performs the search operation in the RODB tree structures, and processes the evaluator code in both parts of the RODB and at the both levels. The Query Thread waits for requests from the local scheduler or the external process that serves requests from other nodes.

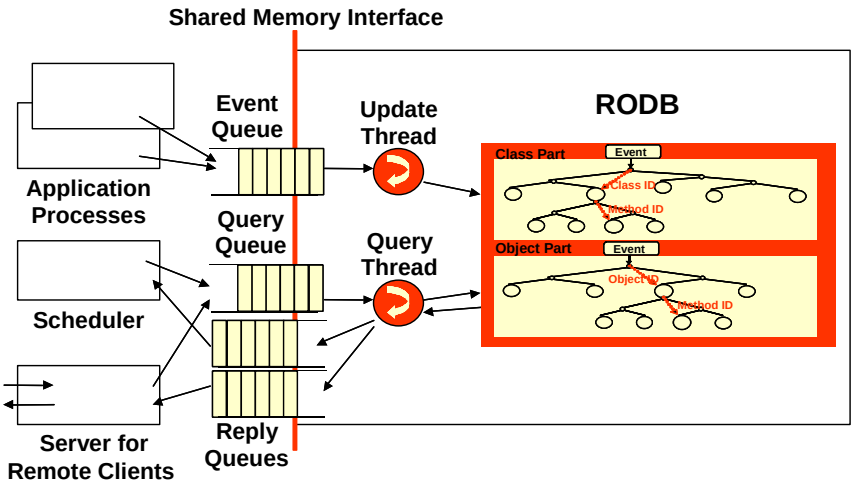


Figure 3.15:Internal structure of the RODB server process

Event and query processing consumes cycles and storing the required information in the RODB requires additional memory capacity. As these resources are usually critical in a real-time system, it is design-goal of the overall system to minimize this resource-consumption and to use non-critical resources (e.g. CPU idle-cycles) whenever possible. As the update of the RODB is not highly time-critical in itself, it can be done asynchronously, e.g. after a monitored activity has terminated the top-level method. The only requirement is, that the database is "reasonably" up-to-date to avoid providing completely out-dated performance figures to the other components.

3.3.2 Online Statistics

The RODB provides the infrastructure for mapping events to classes, objects, and methods. It is completely generic concerning the concrete algorithms used inside the evaluators. Evaluators are consuming events, they are responsible for accumulating any required state, and they finally provide the results of a statistical analysis when requested via the Query Interface. Any evaluator in the RODB is an implementation of the abstract base class *TStatEval*, which provides the necessary recording- and reading-interface to represent its actual statistical evaluation functionality and to report a class-specific set of statistical data. Different evaluators can be linked in a chain. Thus, the system can homogeneously handle different types of evaluators in a common way.

```
template <class t_SampleValue> class TStatEval
// Abstract base-class for evaluators.
{
```

```
  public:

   virtual int Record (const t_SampleValue &NewET) = 0;
   // Records new ET to statistical evaluation.
   // Return value reflects if execution was successful.

   virtual int GetEvaluation (TStatResult &EvalResult) = 0;
   // Updates 'EvalResult' according to the current
   // statistical evaluation data.
   // Return value reflects if execution was successful.
   // (Implementation of elaborated class should be such it can
   // fill on object of the corresponding elaboration
   // of 'TStatResult'.)

   virtual void Reset () = 0;
   // Resets statistics (as if newly constructed).
   // (Does not affect if object is enabled or disabled.)

   virtual void Disable () = 0;
   // Advises statistical evaluation object to ignore
   // incoming samples.

   virtual void Enable () = 0;
   // Advises statistical evaluation object to process
   // incoming samples.

   virtual int Save (ostream &SaveStream);
   // Saves evaluation data to 'SaveStream'.
   // Return value reflects if execution was successful.

   virtual int Load (istream &LoadStream);
   // Loads evaluation data from 'LoadStream'.
   // Return value reflects if execution was successful.

   virtual t_SE_ErrorCode CheckError () const = 0;
   // Returns code of last error that occurred since last call of
   // 'ClearErrorStatus()' (or since creation).

   virtual void ClearError () = 0;
   // Reset internally stored error information to a state
   // as if no error had occurred.

   virtual t_StatType GetEvalType () const = 0;
   // Returns type of statistical evaluation.
   // (Has to match corresponding
   // 'TStatResult'-elaboration's 'GetEvalType()'.)

   virtual TStatEval *MakeNew () const = 0;
   // Returns pointer to a newly created object instance
   // of the elaborated statistical evaluation class.
   // (NULL, if allocation failed.

   virtual ~TStatEval () {};
   // Destroys statistical evaluation object.
};
```

Figure 3.16: Abstract base-class for evaluators

The most important methods of the *TstatEval* class, shown in Figure 3.16, are `Record()` and `GetEvaluation()`. The fist is used to insert an event into the evaluator. Note, that the class is parameterized by *t_SampleValue* – that is the type of the inserted event. This allows for adapting the RODB and the evaluator to new event-formats containing additional or even completely different information. The only constant information that has to remain in an event data record are event identifiers that can be mapped to class and method identifiers and the object identifier. Otherwise routing in the RODB becomes impossible.

The second method, `GetEvaluation()`, is used for extracting analysis results from an evaluator. It operates on an object of the *TStatResult* base-class, which will be described in the next subsection. Depending on the actual sub-type of the evaluator, the sub-type of the result object may differ. The correct type can be determined at runtime using `GetEvalType()` that returns a matching type identifier.

The remaining methods are specifying the API for creating and retrieving a persistent image of an evaluator, for error handling, for enabling and disabling the event consumption (useful for objects that just serve as templates in level zero or one of the RODB and do no processing themselves), and the method `MakeNew()` that clones new instances of the current object.

In order to provide *ECET*s as described in subsection 3.2.1, three different evaluators have been implemented, one that assumes a normal distribution with mean and variance and two that use the discrete density representation, one with the full history of the recent *n* events and one with the negative exponential fade-out algorithm. Performance figures for these evaluators will be given in subsection 3.4.

3.3.3 Query Interface

In order to make use of the progressing analysis, queries have to be sent to the Online Statistics components, the evaluators, in the RODB. Every querying application is a client of the server; this can be any application, like a dedicated scheduler, or even the monitored application itself. In the latter case the online monitoring infrastructure acts again as a tool for providing time-awareness, or in other words it can be seen as an extension to the object-oriented language that adds a new dimension of reflection, namely ”performance reflection”.

To query the server, a query request is put into a local shared memory buffer for the server, similar to the event-recording interface. The results of a query in turn are also passed to the client via shared memory. However, while for queries the server maintains one buffer, which collects the requests of all clients, for receiving the results the client opens an individual shared memory buffer, referred to as *response channel* here. Results in a response channel will always appear in the same order as the requests have been posted. In querying, apart from the re-

quested context, which specifies the object instance and method, the type of evaluator to be queried has to be specified by the client. To select the requested evaluator type, simply an arbitrary instance of the desired class has to be specified. In addition, the request can contain arguments to the evaluator, which influence as parameters the computation of the result.

In Figure 3.17 the declaration of the *TstatResult* class, the base-class for all evaluator results is given. It provides method-templates for accessing the opaque data-structure (that is actually transferred from the RODB server via shared memory and/or the network interface), for printing the data in a human-readable format, for determining the type (in order to match it with the evaluator), and two methods to access the validity of the contained result.

```
class TStatResult
// Abstract base-class for results of 'TStatEval' objects.
{
public:

  virtual void MakeInvalid () = 0;
  // Marks result as being invalid (internally).

  virtual int CheckValidity () const = 0;
  // Returns if result object is containing valid data.

  virtual int TextCopy (char * const StringBuffer,
     const size_t BufferSize,
     const char * const IndentStr = NULL) const = 0;
  // Copies string containing a textual representation of the
  // result into 'StringBuffer' (for human-readable output).

  virtual t_StatType GetEvalType () const = 0;
  // Returns type of evaluation result.
  // (Has to match corresponding 'TStatEval'-elaboration's
  // 'GetEvalType()'.)

  virtual void *DeliverData () = 0;
  // Returns pointer to internal data structure.

  virtual unsigned long GetDataSize () const = 0;
  // Returns size of internal data structure.

  virtual ~TStatResult () {};
  // Destroys result object.

};
```

Figure 3.17: Abstract base-class for results

The details of the query API of the RODB are sketched in terms of a short code sample (see Figure 3.18). In the example the $ECET_{(o,m),k,n}$ is requested from an exponential fade-out evaluator. The parameters *o* and *m* are set by the requesting program as well as the ratio *k*/*n* (in the example 95%). As *n* is a parameter of the evaluator itself, the required *k* is determined during the query. After sending the

query, the result is received from the corresponding response channel. The `ReceiveEvalResult()` method waits until the result is present in the channel or the timeout expires. A timeout value of -1 means immediate return, so it can be used for non-blocking requests, where the result can be collected later from the return channel.

```
#include "RODB_cl.hh"
  ...
  t_RODB_Client     RODB_Client;
  t_ResponseChannel BackChannel;
  TFadingDist       EvalXY;
  // the 'exponential fade-out' evaluator

  TFadingResult     ResultECET;
  // the corresponding result object

  t_ObjectID        ObjectID = o;
  // some object of an instrumented class
  t_MethodID        MethodID = m;
  // the ID of the requested method
  ...

  RODB_Client.OpenResponseChannel(BackChannel);

  ResultECET.SetSamplePercentage(95);
   // the request's result will report on
   // ECET(o, m, 95)
   // the ECET which is equal or greater
   // than at least 95% of the recorded samples

  RODB_Client.RequestEvaluation(
            ResultECET, ObjectID,
            MethodID, ResponseChannel));

  if (RODB_Client.ReceiveEvalResult(ResultXY,
            ResponseChannel,Timeout)) {
     ExpectedTiming = ResultECET.Prediction
      // Now we can use the ECET
      ...
     }
}
```

Figure 3.18: Example of an RODB query

3.3.4 Activity Manager

The second component that receives event data from the Event-Processing component is the Activity Manager. The Activity Manager is synchronously invoked when new events are detected that potentially report on a new mapping of activities to threads (i.e. invocation/return events from the middleware layer).

Even simple tracing of activities in a distributed o-o system is a non-trivial task, as an activity may cross node boundaries several times. In addition, as stated

before, many middleware systems, including standard CORBA, do not even have a build-in activity abstraction that would allow identifying an activity across nodes e.g. by a global ID. The Activity Manager also requires knowledge about the timing requirements associated with an activity. Therefore, it has to implement a mechanism to piggyback this information on the invocations. In order to do this transparently to the application, the Activity Manager closely cooperates with the local instrumentation code that is hooked into each remote method invocation. The timing data and the activity ID mapping to the local threads is maintained by the Activity Manager and then added and extracted to and from each invocation message by the instrumentation code. If required the activity manager can report the current mapping of thread to activities to scheduler (in order to provide it with the knowledge about timing parameters of activities or to enable early detection of timing faults).

3.4 Measurements and Evaluation

With the implementation of the complete monitoring system, a number of measurements have been made in order to give an estimation for the actual overhead introduced by online monitoring and for determining the possible granularity of observation. All given figures have be measured on a Sun UltraSparc 1 running the SunOS 5.5 (Solaris) operating system [Ger99a].

Sensor Performance

For the measurement of the overhead introduced by the instrumentation, code instrumentation with mc4p has been applied. A sub-class of mc4p's *_instrumented_class* has been implemented that reduces the number of events to one per method invocation. On the start of a method it stores the timestamp in a local variable and at the end it reports the time difference directly in one event.

The interference introduced by the execution of this sensor code has been measured. The overhead includes the time for taking two 64-bit timestamps (at the beginning and at the end of the measured method), for computing the time-difference, locking the event queue, and for enqueuing the event data. The time was measured to be about 5 µsec per sensor. In the current Solaris implementation a major part (30%) of the overhead results from the "gethrtime()" system call that is used to access the nanosecond counter. This can be improved by either using a memory-mapped version of this counter or, on other architectures (e.g. the Intel Pentium), by using on-chip counters. In both cases the overhead reduces dramatically (just one instruction on the Pentium). Another 10% of the sensor overhead is due to the locking of the event queue. As prior mentioned, in some system environments this can be simplified to a plain "disable interrupts" in the single processor case. But even without these optimizations, it can be stated that 1000 events/s can be created with using less than 0.5% of the CPU power of the machine.

RODB Performance

The figures in the previous subsection do not yet include the processing power needed for storing the events in the RODB. Therefore, also the performance of the RODB depending on the size of the object-space and the complexity of the used evaluators has been measured. With the “negative exponential fade-out” type evaluator (which can be considered as having a medium complexity) and an object-space of 100 classes, 500 objects, and 200 methods per class, about 60.000 event/s (~17 µsec/evens) has been achieved. Either by decreasing the object-space by a factor of 100 or by using a very simple statistical evaluator (incremental computation of average and variance) is was possible to speed-up event processing by another 10%. A query into the database has about the same time-complexity as an event input. This means that the RODB uses less than 2% of the computing power of the machine (possibly at idle priority) to process continuously 1000 events/s resulting from 1000 instrumented methods executions per second.

Conclusion

At a first glance 1000 methods per second seem not to be a lot in an up-to-date object oriented system. However, it has to be conceived that for supporting the TAFT-Scheduler, only the monitoring of scheduling-relevant entities is required. This means, that only methods of the size of “tasks” have to be instrumented. For those, a granularity in the order of milliseconds is quite reasonable. Also, if you consider remote invocations in a distributed object-oriented system via a network, like e.g. in CORBA, 1000 methods per second is a realistic order of magnitude. This clearly shows that our online monitoring system can provide the required generic support for the scheduler without needing significant additional CPU resources.

3.5 Case Study – RTL-based Constraint Checking

The presented monitoring and event-processing infrastructure supports the concept of TAFT-Scheduling for an adaptive, object-oriented real-time system. However, the presented instrumentation techniques and also the sensors and the event-infrastructure of the RODB can be used for other purposes in object-oriented real-time systems as well. They can even be useful in systems that have no notion of soft-tasks as required for the TAFT-Scheduling approach.

In this study, a concept is presented that applies the techniques of this thesis to hard real-time systems in order to implement an object-oriented checker for timing-constraints written in the RTL formal language [Ger96b]. The first subsection shortly describes the existing formalism and its extension for object-oriented languages. The second part explains how the tools presented in the previous chapters can be applied to implement this checker.

3.5.1 Event-based Timing Constraints in Objects

In the design of all real-time systems, assumptions about the behavior of the system and its environment are made. Assumptions may be related to the external world (e.g. a maximum event rate), to the hardware components (e.g. sensor response times), or to the software itself (e.g. worst case execution times of routines). In the best case these assumptions are based on a formal analysis. But even a formal treatment of a problem has to be based on a certain system model that does not necessarily cover all relevant aspects of the system (e.g. faulty components). Traditionally, a real-time system relies completely on the correctness of its design assumptions and the system's behavior becomes undefined as soon as an assumption is violated. In many modern applications this is not acceptable any more and so fault-tolerance has become another major issue in the design of real-time systems. The first step in fault-tolerance is fault-detection. In order to react on the violation of design assumptions, the system software (either the operating system or the application) has to be informed when such a violation occurs. Thus, runtime-checks of the system's behavior are an integral part of a fault-tolerant real-time system.

Major work in the area of online checking of timing constraints with an event-based system has been done Jahanian al. As already discussed in the related work in subsection 3.1.5, their RTL-based constraint checkers [Cho91, Raj92, Jah94] use a model of monitoring that is comparable to the one presented in this thesis. However, in their work no assumptions are made how this fits together with object-orientation, how the work can be integrated with standard programming languages, and how the code can be instrumented with event-triggers. The concept presented here extends the work towards object-oriented real-time systems. It focuses on how object-orientation can be utilized to simplify the specification and the checking of timing constraints and how this can be integrated into an existing programming language, namely C++.

Specifying and Checking Timing Constraints

It seems to be attractive to add a notion of "timing-abstraction" to the object's interfaces. Similar to data-abstraction the "timing-abstraction" can define the temporal behavior of an object independent of it's actual implementation. Like in non-object-oriented real-time software, timing can be expressed either in a specificational notation or in a more constructive manner.

A specificational notation (like RTL) allows to specify complex system behavior including the behavior of environmental components, but it does not show a way how to really implement the behavior in a real system. An approach to utilize a specification after the design and verification phase of a software project is a runtime system that allows to check the behavior of the actual application against the timing constraints written in a formal language. In case of a detected timing constraint violation an exception-condition is raised. In response to exception application-specific error recovery mechanisms may be triggered. These recov-

ery mechanisms may range from a complete shut-down to the activation of hot-standby resources. Such an "checker"-approach is comparable to the definition of pre- and post-conditions that also help to check the behavior of an application but do not propose any solution to the problem of how to implement the required properties.

In an object-oriented environment it seems to be quite natural to specify the timing in the same place where also the functional description is given: in the class description. This means, that all objects of one class have the same functional and timing behavior. If an object is needed with an identical functional behavior and different timing constraints, a new class has to be defined. This is equivalent to the situation where parts of the functional behavior have to be changed. In both cases inheritance should enable code-reuse, so that only these parts that affected by the changes have to be redefined. An alternative approach is to define the temporal and the functional behavior in different class hierarchies and to combine two classes from both hierarchies to create an instance of an object. This separation avoids extensive re-definitions of functional code if only the constraints have been changed (an effect known as the *inheritance anomaly* [Mat93]), but it introduces redundancy and a source of inconsistency. To avoid this, the concept proposes an approach where the timing constraints are included in the class definition. As the RTL-style rules are independent of the functional class code, changes in these rules cannot lead to redundant re-definitions of class methods.

The RTL Language

RTL is a formal language for reasoning about timing properties of real-time systems. It describes the absolute timing of events (not just the ordering of events). An event marks a point in time, which is of significance to the behavior of the real-time system. RTL distinguishes three types of events: external events coming from the environment, start/stop events marking the beginning/end of an execution of a code sequence, and transition events that indicate changes of certain variables of the system. Events have unique names. Time is captured by the occurrence function @:Event x Instance $\rightarrow$ Time that assigns a time value to an event occurrence. The expression $@(X,i)$ denotes for example the time-stamp of the i^{th} occurrence of the event X. RTL formulas are constructed using addition and subtraction of occurrence functions with integers, (in)equality predicates, universal and existential quantifiers, and the first-order logic connectives.

The automatic checking of safety assertions about a system with a specification written in these general RTL-formulas is extremely inefficient and can only be used for small systems. But a similar formalism can also be used to check an actual run of a system (a sequence of events) against its specification. Three subsets of RTL have been defined in [Jah90] which can be evaluated in polynomial time to decide even for an incomplete computation whether it still can fulfill its specification or not. The most important subset are those formulas that only

consist of occurrence functions with a constant occurrence index. A positive index value *i* denotes the absolute number of the occurrence (e.g. the first), while a negative value specifies an occurrence relative to the current point in time (i.e. @(*X*,-1) is the time when the last instance of event *X* has happened). The papers [Cho91] and [Raj92] describe a graph-based algorithm that evaluates these formulas, whenever a new event arrives or when a timeout expires. In addition they defined a new access function @val: Variable x Instance → Value for the use in timing constraints. The function results in the value of a variable after the i^{th} occurrence of a transition event for this variable.

Basic Events

The object-oriented extension uses the same basic events as already known from RTL: "Start" and "End" events of code sections and state changes of variables. In the case of object-oriented system this matches with the beginning and the end of methods and the changes of member variables. All these basic events are implicitly defined by the definition of the corresponding member components. An event consists of four components: the static event name, the dynamic context, a time-stamp, and additional optional parameters.

The static event name denotes the static context in which the event occurred. In case of a basic event the static event name is expressed by the name of the class and either the name of the method (plus "start" or "end) or the name of the member variable. The dynamic context of the event is the unique object-identifier of the producing object. Using a clock synchronization, e.g. as the a-posteriori algorithm as described in subsection 2.2.2, it is assumed that the time-stamps impose a global order on the events of the distributed. The fourth component holds additional optional parameters, like the new value associated with a state change event.

Timing Constraints

In order to specify the timing of objects a new component is added to a class description: the `constraint` section. This section contains a list of named RTL-like formulas. The formulas are composed out of the basic events as described above. A constraint itself also defines an event. When a constraint is violated, the object produces an event with the static name of the constraint, the dynamic context of the object that violated the constraint and a time-stamp that is the earliest point in time when the checker could evaluate that the constraint will be violated. This allows to easily composing more complex events out of basic events. In our current approach events are not first-class objects. They cannot be used in the functional specification of the class (in the "normal" C++ part) and there exist no variables of type "event". This independence of functional specification and timing specification avoids inheritance anomalies and it allows the construction of two separate systems: the object oriented real-time system and its constraint checker.

A simple example of a constraint list is shown in Figure 3.19. Constraint 1, named "max_time", states that an execution of the method `get_val()` must not take longer than 4 ms. The expression `@(get_val.start,-1)` denotes the start time of the most recent execution of the method `get_val()` and @(get_val.end,-1) evaluates to the end time of the same execution. Constraint 2, named "recovery", expresses that two successive calls to `get_val()` have to have a distance of at least 1 s.

In a similar manner a period, a jitter, or a time-out can be defined. The expressiveness of RTL-like formulas also allows to define constraints between the execution of different methods and rules that depend on the value of member variables. This enables to write rules that define synchronization conditions and mode-changes. In order to simplify the writing of rules without loosing the complete expressiveness of the constraint formulas the standard C++ macro-mechanism can be used. Figure 3.20 shows how the `max_time` constraint from the example above can be simplified using a standard C++-macro for defining a deadline.

```
   class sensor {
public:
       int get_val();
       :
       [[// The Constraint Section
       // Constraint 1:
       // get_val() must not take longer than 4 ms
              max_time:    @(get_val.start,-1) >=
                           @(get_val.end,-1) - 4ms;
       // Constraint 2:
       // get_val() must not be called more that once per
second
              recovery:    @(get_val.start,-2) <=
                           @(get_val.end,-1) - 1s;
       ]]
}
```

Figure 3.19: A C++ class with timing constraints

```
#define deadline(func,time) (@(func.start,-1) >=  \\
                             @(func.end,-1) - time)
       ...
       // Constraint 1:
       // get_val() must not take longer than 4 ms
              max_time: deadline(get_val, 4ms);
       ...
```

Figure 3.20: The usage of macros to simplify the notation

Note, that all these constraints are local to an object. The constraints are checked on a per object basis and they become active, if an object violates one of them.

This implies that for every new object also new instances of the events and new instances of the constraints are created. This is a major difference to existing event-based constraint checking tools. Code sharing is a standard feature in object-oriented languages. This implies that the event-producing code is also shared. If an event is only determined by the location of its sensor and not by its producing object, all objects of the same class would produce the same events. Based on these events all timing constrains would be evaluated on a per class basis and this is in most cases not the desired semantics.

The timing constraints described so far are also local in a sense that they only refer to events that are produced by the execution of the object itself. But this kind of constraints is not sufficient. They can guarantee the local consistency of objects, but cannot capture inter-object dependencies in the system. This requires constraints that combine events from different objects. To achieve this, objects have to export events via their interface. But as "timing abstraction" was one of the goals of our approach, it is not desirable to export all events of the interface of an object, but only those that are independent of the implementation and relevant to others. This is marked in C++ style with the keywords `public`, `protected`, and `private` that determine the visibility of an object components. `Public` events are visible globally, while `protected` events are only meaningful within the inheritance hierarchy. `Private` events are purely class internal. Due to the limitations of the static analysis, inter-object constraints are limited to components of an object where the references are known at compilation time. This also means, that constraints can be checked locally at each node as components of an object are considered to be always located on the same node as the containing object.

Figure 3.21 shows an example of a class that has timing constraints based on local basic events and on events that are exported by some of its components. E.g. the expression @(left.max_time,-1) denotes the most recent time, when the constraint `max_time` in the object "left" has been violated. This time is defined by the constraint-checking algorithm. The algorithms described in [Jah94] guarantee, that a violation is detected at the earliest possible time. If no occurrence of e.g. left.deadline ever happened, the expression @(left.max_time,-1) is treated to denote some time-value in the future. Thus, according to the algorithm given in [Jah90], a constraint like @(A,1) < @(B,1) ("A happens before B") is already known to be violated at the moment when B happens first, as no value of @(a,1) can ever fulfill the constraint. There is no need to wait with the evaluation of the formula until A really happens. The constraint `input_incorrect` of the class `element` states, that the `max_time` constraint must not be violated by one of the sensor objects (named `left` and `right`) while the method `get_position()` is executed.

```
class element {
sensor right, left;
       get_position() {
              int l = left.get_val();
              int r = right.get_val();
              ...};
       :
       [[
       // Constraint 1:
       // a deadline violation of one of the
       // sensors must not happen
       // while the position is determined
              input_incorrect:
                                  @(get_position.start,-1) >=
                                  @(left.max_time,-1) >
                                  @(get_position.end,-1)
                                  ||
                           @(get_position.start,-1) >=
                                  @(right.max_time,-1) >
                                  @(get_position.end,-1);
       :
       ]]
}
```

Figure 3.21: A C++ class with inter-object timing constraints

Inheritance

In an object-oriented environment the semantics of events has to describe what happens to events in an inheritance hierarchy. Events and constraints are inherited like other members of classes. If an event is produced by a base-class the same event will be produced by any derived class as well. The same holds for constraints. Events and constraints can also be overwritten. Basic events are redefined by overwriting the defining method or member variables, while constraints (and the according violation events) are redefined by declaring a new constraint with the same name. Overwriting a constraint with an empty constraint disables it.

The binding of events in constraints is dynamic in sense that events are always tested with the constraints that belong to the actual class of the producing object, not in the context of the class that first defined this event.

This kind of inheritance allows to redefine timing constraints of a class by inheriting the functional code and redefining its timing. Of course, this is also possible for classes that had no timing constraints before, like classes from an existing C++ application.

3.5.2 Infrastructure for Constraint Checking

The actual checking of the constraints defined in the C++ like language requires a compile- and runtime support that is very similar to the requirements of the ECET-prediction in the TAFT-Scheduler.

The Compiler

A compiler for the proposed language extension has to do two additional tasks besides the production of the object code. It has to translate the timing constraints into a) an instrumentation of the object-oriented program in order to produce the required events and b) a representation of the constraints that can be evaluated by the constraint checker. The task of adding the instrumentation can be done by a mc4p as described in section 2.3.2. The events that identified by mc4p are exactly the basic events as defined above.

For the second task the constraints have to be parsed and converted into a representation that is suitable for efficient constraint checking. As described in [Cho91] this can be done by converting the formula into disjunctive normal form, where each basic predicate is an inequality of the form

@(A, i) <= @(B, j) - C (A and B are events; i, j, and C are integer constants)

and finally by converting each disjunction into a graph-template that reflects the dependencies between the event occurrences. These graph-templates have a vertex for every occurrence function and directed edges weighted with the times -C (see Figure 3.22).

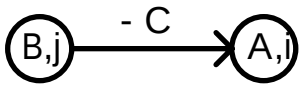


Figure 3.22: Graph-template for @(A, i) <= @(B, j) - C

The Online Checker

The checker has to receive the static information about the structure of the timing constraints from the compiler (see Figure 3.22). This information includes the class structure, the per class event table, the graph-templates representing the temporal dependencies between the events as stated by the constraints. During runtime the checker has to react on incoming events. Upon the reception of a creation/deletion event it has to create/delete an instance of the dynamic object-specific data-structures. These data structures hold all the necessary information to check the timing of this particular object. Upon reception of such an event these constraints have to be checked again.

A constraint violation is detected by constructing a current instance of a graph out of the graph-templates (algorithm from [Cho91]). The algorithms works like

this: For each edge with weight *C* in the graph-template that leads to (comes from) a vertex with an already assigned value *T*, an edge in the new graph with weight *C-T* (*C+T*) leading to (coming from) a special "zero-node" is created. Nodes that have not yet an assigned value (the event did not happen), are connected to the zero-node via an edge with the weight -NOW (where NOW is the evaluation time), indicating, that the event might happen in the future. If there is any negative cycle in the graph the disjunction is unsatisfiable. If all disjunction in the DNF of a constraint are unsatisfiable, the constraints has been violated

Upon detection of a constraint violation the checker itself produces the according event that then will be immediately checked (it is by definition the next event in the total order of events). For deadline-like constraints the checker also inserts special time-out events into the queue, in order to detect a possible violation as early as possible. As events are queued according to their time-stamps in front of the checker, this does not necessarily mean, that a violation is detected when it actually happens, but it is detected before any more recent event is processed.

The RODB reflects pretty much the required structure of the constraint checker. Each object has its separate evaluator that now executes the graph-based checking algorithm. Upon reception of a new event that belongs to a specific objects it is routed to the right evaluator using the same mechanisms as described above in subsection 3.3.1. Basically one additional mechanism is required. For the evaluation of inter-object constraints, event have to be forwarded to the right consumer object's checker. This can be accomplished by a list of object references maintained by each evaluator, where the interested consumers register themselves online during object creation.

There are three possible modes in which a checker can be used: off-line, online, and real-time. In the off-line mode performance of the checker is not an issue as long as it is possible to get results about an execution after a reasonable time of waiting. An online checker has to be able to cope with the average event rate, so that it can keep track with the running system. This is probably enough to provide feedback for an adaptation mechanism. If the checker itself runs as part of the real-time system, it can provide direct input for the decisions of the application. This would require that parts of the system described so far must have known worst case execution times and that they are scheduled with the application itself. This is true for the sensor part but it has not been a design-goal of the RODB.

Conclusion

The mechanisms described for instrumentation and event processing in object-oriented real-time systems can be used for monitoring and visualization. In a second step the same data can be used for online analysis and a feedback into the system's runtime, if the runtime has the mechanisms to exploit this additional knowledge. But these are not the only applications for object-specific timing-data. The presented online checker extends previous ideas about online evalua-

tion of formal constraints in the running system towards object-oriented systems. It is not only that this object-oriented approach allows for more fine-granular constraints, but it is the prerequisite to apply such an event-based algorithm in an object-based environment. Classic, static code-based event handling must fail because of the use of code sharing. The presented RTL-based checking algorithm can be seen as an example. If other algorithms are more adequate for a concrete target system, the evaluator-based structure of the RODB provides a generic interface for inserting any appropriate checking/evaluation module.

4 Summary

The traditional view of real-time systems as isolated, embedded systems does not longer suffice for future complex open control systems. The use of the object-oriented paradigm has already been accepted as a design methodology for real-time systems that greatly reduces the complexity of the system while improving reusability and manageability. As also the surrounding IT-infrastructure is more and more accessible through object-oriented interfaces, this directly suggests to use object-orientation as the integrating communication paradigm in these open heterogeneous systems. However, as CORBA, DCOM and comparable object-oriented middleware and also most of the applications running on top are not aware of real-time requirements, a serious problem arises. The evident approach to develop a real-time capable object-oriented runtime system and to implement the complete application in a homogeneous real-time environment is usually not a practicable solution, as the involved applications are not designed for real-time requirements. Moreover, a main concept of object-oriented, namely implementation hiding, collides with the need of typical real-time systems for total knowledge and control of the required resources. This dilemma results in a separation of object-oriented systems into a real-time and a none-real-time domain. Preserving the heterogeneity of these domains and providing appropriate mechanisms for interfacing non-real-time and real-time objects, is probably the only viable approach to tackle this problem.

A first step towards a successful interoperability of these domains is time-awareness, i.e. the ability of the systems to monitor, gather information, and report about its own timing behavior. The availability of this information is the necessary precondition that a system can be operated in a time-critical environment. For a complete view, monitoring must happen at all architectural levels and it should preserve and exploit the structural information provided by object-orientation. As this is a generic job for all applications interfacing to the real-time domain, it can and should be supported by system infrastructure. This thesis presented instrumentation concepts for the operating system, the middleware, and the language level and tools for a distributed environment that combine the gathered information in a novel and for a real-time system designer intuitive way.

However, in order to give guarantees for real-time behavior in a heterogeneous environment with non-real-time and real-time objects, gathering timing information of the system it not enough. Resources have to be managed according to the acquired knowledge, a scheduling problem. Traditionally real-timer schedulers are based on the notion of *WCET*s, but this concept is increasingly inappropriate for the considered systems. In order to cover the degree of unpredictability the concept of *ECET*s has been introduced, that expresses estimates instead of upper bounds for the future timing behavior. ECETs can be computed by monitoring and extrapolating the timing behavior of individual objects from the near past

into the near future. The knowledge of ECETs can be used by a dynamic scheduler to adapt its resource allocation decisions depending on the current state of the system and the environment. It also enables the objects in the real-time domain to anticipate the behavior of invocations of non-real-time objects and to react accordingly. The thesis described the design and the implementation of an integrated execution time prediction infrastructure that is able to compute ECETs efficiently during runtime. The implementation of the system exhibits performance figures that proof the viability of the approach. The results enable future object-oriented real-time systems to use the ECET-knowledge to minimize the unpredictability when invoking other services and enable them to establish efficient timing-fault handling that ensures computational progress even in overload situations.

5 References

[Arn94] Arnold, R., F. Mueller, D. Whalley, and M. Harmon, Bounding Worst-Case Instruction Cache Performance, in Proceedings of the 15th Real-Time Systems Symposium, pp. 172-181, 1994.

[Bas94] Basumallick, S., K. Nilsen, Cache Issues in Real-Time Systems, ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems, Jun. 1994.

[Bec00] Becker, L.B. and C.E. Pereira, From Design to Implementation: Tool Support for the Development of Object-Oriented Distributed Real-Time Systems, in Proc. of 12th Euromicro Conference on Real-Time Systems, Stokholm, Sweeden, pp. 108-115, June 2000.

[Bec99] Becker, L. B., Gergeleit, M., Nett, E., Pereira, ., C. E., An Integrated Environment for the Complete Development Cycle of an Object-Oriented Distributed Real-Time System, 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'99), Saint-Malo, France, pp. 165-171, May 1999.

[Bih91] Bihari, T. E. and Schwan, K., Dynamic adaptation of real-time software, ACM Transactions on Computer Systems, 9(2), pp. 143-174, 1991.

[Bol00] Bollela, G. et al., The Real-Time Java for Java Experts Group, Addison Wesley, Reading, MA, 2000.

[Boo91] G. Booch, Object-Oriented Design with Applications, Benjamin/Cummings Publishing, Redwood City, CA, USA, 1991

[Boo99] Grady Booch, James Rumbaugh, and Ivar Jacobson, The Unified Modeling Language User Guide, Addison Wesley, Reading, MA, 1999.

[But97] Buttazzo, G. C., Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, The Kluwer International Series in Engineering and Computer Science, Kluwer Academic Publishers, 1997.

[Cah93] Cahill, V., Balter, R., Harris, N.R., Rousset de Pina, X. (Eds.), The COMANDOS Distributed Application Platform, Research Report ESPRIT Project 2071, Springer, 1993.

[Cho91] Chodrow, S. E., Jahanian, F., Donner, M., Run-Time Monitoring of Real-Time Systems, Proc. of Real-Time Systems Symposium, San Antonio, Texas, pp. 74-83, Dec. 1991.

[Cus93] Custer, H., Inside Windows NT, Microsoft Press, Redmond, WA, USA, 1993.

[Dod92] Dodd, P. S., Ravishankar, C. V., Monitoring and debugging distributed real-time programs, Software Practice and Experience, Vol. 22, Nr. 10, pp. 863-877, Oct. 1992.

[Dou98] Douglass, Bruce Powel, Real-Time UML: Developing Efficient Objects for Embedded Systems, Addison-Wesley, Reading, MA, 1998.

[Edd99] Eddon, G., Eddon, H., Inside Distributed COM, Microsoft Press, Redmond, WA, USA, 1998.

[Foo99] Foote, W., Real-Time extensions to the Java™ Platform - A Progress Report, Proc. Fourth International Workshop on Object-oriented Real-Time Dependable Systems (WORDS'99), Santa Barbara CA, 1999.

[Ger92] Gergeleit, M., F. Lange, R. Kröger. Microkernel Performance Evaluation using the JEWEL Distributed Measurement System", Proc. OpenForum'92, Technical Conference, Utrecht, Niederlande, pp. 219-231 Nov. 1992.

[Ger94] Gergeleit, M., Automatic Instrumentation of Object-Oriented Programs, Arbeitspapiere der GMD, Nr. 826, Feb. 1994.

[Ger95] Gergeleit, M. and H. Streich. Synchronizing High-Resolution Clocks via the CAN-Bus, 28th ISATA Dedicated Conference on Mechatronics - Efficient Support for Engineering, Manufacturing, Testing & Reliability, Stuttgart, Sep. 1995.

[Ger96a] Gergeleit, M. and H. Streich. TaskPair-Scheduling with Optimistic Case Execution Times - An Example for an Adaptive Real-Time System, Second International Workshop on Object-oriented Real-time Dependable Systems (WORDS96), Laguna Beach, California, Feb. 1996.

[Ger96b] Gergeleit, M., J. Kaiser, H. Streich, Checking Timing Constraints in Distributed Object-Oriented Programs, OOPS Messenger, ACM Press, Vol. 7, No. 1, pp. 51-58, New York, NY, Jan. 1996.

[Ger97a] Gergeleit, M., M. Mock, E. Nett, J. Reumann. Integrating Time-Aware CORBA Objects into O-O Real-Time Computations, Third International Workshop on Object-oriented Real-time Dependable Systems, Newport Beach, Ca., USA, Feb. 1997

[Ger97b] Gergeleit, M. and M. Mock, Real-Time Monitoring of the EIVIS Distributed Video-Server on Windows NT, 18th IEEE Real-Time

Systems Symposium, Work in Progress, (RTSS-97), San Francisco, Ca, USA. Dec. 1997.

[Ger98] Gergeleit, M., M. Mock, E. Nett, T-CORBA: Making Object-Oriented Systems Time-Aware, Computer Systems Science & Engineering, Vol 13, No 3, pp. 151-160, May 1998.

[Ger99a] M. Gergeleit, J. Fitzner. Using Structure-Based Measurements for Predicting Execution Times of Object-Oriented Programs, International Journal of Parallel and Distributed Systems & Networks, Vol 2, No. 3, ACTA Press, Calgary, pp. 118-126, 1999.

[Ger99b] Gergeleit, M., E. Nett. JewelNT: Monitoring of Distributed Real-Time Applications on Windows NT, Proc. 3rd Annual IASTED International Conference on Software Engineering and Applications (SEA'99), Scottsdale, AZ, USA, pp. 325-328, Oct. 1999.

[Ghe93] Gheith, T. and K. Schwan, CHAOS-Arc - Kernel Support for Multi-Weight Objects, Invocations, and Atomicity in Real-Time Applications, ACM Transactions on Computer Systems, 11(1), pp. 33-72, 1993.

[Gra82] Graham, S. L., Kessler, P. B., McKusick, M. K., gprof: a call graph execution profiler, in Proc. SIGPLAN'82 Symp. Compiler Construction, pp. 120-126, 1982.

[Hab89] Haban and D. Wybranietz, Behavior and Performance Analysis of Distributed Systems Using a Hybrid Monitor, International Computer Science Institute, Tech. Rep. TR-89-029, Berkeley, CA, May 1989.

[Hab90] Haban, D. and K.G. Shin, Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times, IEEE Transactions of Software Engineering, 16(12), pp. 1374-1389, 1990.

[Hea94] Healy, C. A., D. B. Whalley, M. G. Harmon, Integrating the Timing Analysis of Pipelining and Instruction Caching, in Proc. IEEE Real-Time Systems Symposium, pp. 288-297, Dec. 1995.

[Ion95] Orbix reference guide, IONA Technologies, 1995, Dublin, Ireland

[Ish90] Y. Ishikawa, H. Tokuda, C.W. Mercer, Object-Oriented Real-Time Language Design: Constructs for Timing Constraints, Proc. ECOOP/OOPSLA '90, Oct. 1990.

[Jah86] Jahanian, F., A. Mok, Safety analysis of timing properties in real-time systems, IEEE Trans. Software Eng., Vol. SE-12, No. 9, pp. 890-904, Sept. 1986.

[Jah87] Jahanian, J. and A. Mok, A Graph-Theoretic Approach for Timing Analysis and its Implementation, IEEE Transactions on Computers, Vol. C-36, No. 8, Aug. 1987.

[Jah90] Jahanian, F. and A. Goyal, A formalism or Monitoring Real-Time Constraints at Run-time, in Proc.IEEE Fault-Tolerant Computing Symp., pp. 148-155, June 1990.

[Jah94] Jahanian, F., R. Rajkumar and S. Raju, Runtime Monitoring of Timing Constraints in Distributed Real-Time-Systems, Real-Time-Systems, Vol. 7, No. 3, pp. 247-274, Nov. 1994.

[Jai91] Jain, R., The Art of Computer Systems Performance Analysis, John Wiley & Sons, New York, USA, 1991.

[Jen85] Jensen E. D., C. D. Locke, and H. Toduda, A time-driven scheduling model for real-time operating systems, in Proceedings of the IEEE Real-Time Systems Symposium, pp. 112-122, 1985.

[Kai99] Kaiser, J. and Nett, E., Echtzeitverhalten in dynamischen, verteilten Systemen, in: Informatik Spektrum Nr. 21, Gesellschaft für Informatik eV, Springer Verlag Berlin, Heideberg, 1999.

[Ken94] Kenny, K. B. and K.-J. Lin, Building Flexible Real-Time Systems Using the Flex Language, Computer 24:5, May 1991.

[Kim99] Kim, S.-K., S. L. Min, and R. Ha, Analysis of the Impacts of Overestimation Sources on the Accuracy of Worst Case Timing Analysis, in Proceedings of the 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, pp. 22-31, 1-3 Dec. 1999.

[Kli86] Klingerman, E. and A. Stoyenko, Real-Time Euclid: A Language for Reliable Real-Time Systems. IEEE Transactions on Software Engineering, 12(9), pp. 941-989, Sept. 1986.

[Kop97] Kopetz, H., Real-Time Systems, Kluwer Academic Publishers, 1997.

[Kri97] Krishna, C. M and K. G. Shin, Real-Time Systems, McGraw-Hill Series in Computer Science, 1997.

[Kru98] Krupp, P. et al., Adaptable Real-Time Distributed Object Management for Command and Control Systems: Volume II, MITRE Technical Report 98B0000067, The MITRE Corporation, Bedford, MA, 1998.

[Lan92] Lange, F., R. Kröger, M. Gergeleit. JEWEL: Design and Implementation of a Distributed Measurement System, IEEE Trans. on Parallel and Distributed Systems, Vol. 3, No. 6, pp. 657-671, Nov. 1992.

[Lim94] Lim, S.-S., Y. H. Bea, G. T. Jang, B.-D. Rhee, S. L. Min, Y. C. Park, H. Shin, C. S. Kim, An accurate worst case timing analysis for RISC processors, in IEEE Real-Time Systems Symposium, pp. 97-108, 1994.

[LiM95] Li, Y.-T. S., S. Malik, A. Wolfe, Efficient Microarchitecture Modelling and Path Analysis for Real-Time Software, in Proc. Real-Time Systems Symposium, pp. 298-307, Dec. 1995.

[Lin00] Lindgren, M., H. Hansson, H. anf H. Thane, Using Measurements to Derive the Worst-Case Execution Time, in Proc. of RTCSA 2000 Cheju Island, South Korea. IEEE Computer Society, 2000.

[Lin88] Lin, K.-J. and S. Natarajan, Expressing and Maintaining Timing Constraints in FLEX, in Proceedings of the IEEE Real-Time Symposium, pp. 96-105, 1988.

[Liu94] Liu, J.W.-S., W.-K. Shih, K.-J. Lin, R. Bettati and J.-Y. Chung, Imprecise Computations, Proc. of the IEEE, 82(1), pp. 68-82, 1994.

[Loc94] Lockhart, H., OSF DCE: A Guide to Developing Distributed Applications, McGraw-Hill, Apr. 1994.

[LuS99] Lu, C., J. A. Stankovic, G. Tao, and S. H. Son, The Design and Evaluation of a Feedback Control EDF Scheduling Algorithm, 20th IEEE Real-Time Systems Symposium Phoenix, AZ, USA, Dec.1999.

[Mah01] Mahrenholz, D., Minimal Invasive Monitoring, in proceedings of The Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001), Magdeburg, Germany, 2.-4. Mai 2001

[Mar91a] Marzullo, K. and M. Wood, Making real-time reactive systems reliable, ACM Operating Systems Review, 25(1), pp.45-48, 1991.

[Mar91b] Marzullo, K., K. Birman, R. Cooper, M. Wood, Tools for Monitoring and Controlling Distributed Applications", IEEE Computer, 24(8), pp. 42-51, 1991.

[Mat93] Matsuoka, S. and A. Yonezawa, Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, in Research Directions in Concurrent Object-Oriented Programming, (eds.) G. Agha, P. Wegner and A. Yonezawa, MIT Press, pp. 107-150, 1993.

[Mil86] Miller, B. P., C. Macrander, S. Sechrest, A distributed programs monitor for Berkley UNIX, Software Practice and Experience, Vol. 16, Nr. 2, pp. 206-200, Feb. 1986.

[Moc00] Mock, M., M. Gergeleit, E. Nett. Monitoring Distributed Real-Time Activities in DCOM, 3nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'2K), Newport Beach, Ca., USA, pp. 26-33, Mar. 2000.

[Mos97] Moser L. E., P. M. Melliar-Smith and E. Thomopoulos, Probabilistic analysis of real-time dependable systems, in Proceedings of the 3rd International Workshop on Object-oriented Real-time Dependable Systems, Newport Beach, CA, pp. 306-313, Feb. 1997.

[Net01] Nett, E., M. Gergeleit, M. Mock. Enhancing O-O Middleware to become Time-Aware, Real-Time Systems, 20 (2), Kluwer, pp. 211-228, Mar. 2001.

[Net96] Nett, E., H. Streich, P. Bizzari, A. Bondavalli, F. Tarini, Adaptive Fault-Tolerant Policies with Dynamic Real-Time Garuantees, Proc. WORDS '96, Second Int. Workshop on Object-oriented Real-time Dependable Systems, Laguna Beach, 1996.

[Net97a] Nett, E., Real-Time Behaviour in a Heterogeneous Environment?, in: Pro-ceedings of WORDS'97, Newport Beach, February 5-7, IEEE Computer Soci-ety, Los Alamitos, California, pp. 275-281, 1997.

[Net97b] Nett, E. and M. Gergeleit, Preserving Real-Time Behavior in Dynamic Distributed Systems, Proc. of the Int. Conf. on Intelligent Information Systems, The Bahamas, Dec. 8-10, 1997.

[Net97c] Nett, E. and H. Streich, The GMD-Snake - Real-Time Scheduling of a Flexible Robot Application at Run-Time, Int. Workshop on Parallel Computation and Scheduling in Computers, Ensenada, Mexico, 1997.

[Net98] Nett, E., M. Gergeleit, M. Mock. An Adaptive Approach to Object-Oriented Real-Time Computing, Proc. ISORC'98, Kyoto, Japan, Apr. 1998.

[Nil95] Nilsen, K.D. and B. Rygg, Worst-Case Execution Time Analysis on Modern Processors. in ACM SIGPLAN 1995 Workshop on Languages, Compilers, and Tools for Real-Time Systems, San Diego, USA, 1995.

[OMG01] Response to the OMG RFP for a UML Profile for Schedulability, Performance, and Time – Revised Submission, OMG document number: ad/2001-06-14, OMG, Framingham, MA, USA, June 2001

[OMG95] The Common Object Request Broker Architecture, Revision 2, OMG, 1995.

[OMG99a] OMG Unified Modeling Language Specification, Version 1.3, June 1999, OMG, Framingham, MA, USA.

[OMG99b] Real-Time Corba, Joint Revised Submission, www.omg.org, OMG Document ptc/99-05-03, OMG, Framingham, MA, USA, May 1999.

[Pet99] Petters, S. M. and G. Färber, Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible. In Proc. of the International Conference on Real-Time Computing Systems and Applications, 1999

[Pus89] Puschner P. and C. Koza, Calculating the maximum execution times of realtime programs, Journal of Real-Time Systems, vol. 1, pp. 159-176, 1989.

[Pus98] Puschner P. and R. Nossal, Testing the results of static worst-case execution time analysis, in Proc. IEEE Real-Time Systems Symposium, Madrid, Spain, 1998.

[Rac00] Rackl, G., M. Lindermeier, M. Rudorfer, and B. Süss, MIMO – An Infrastructure for Monitoring and Managing Distributed Middleware Environments. In J. Sventek and G. Coulson, editors, Middleware 2000 – IFIP/ACM International Conference on Distributed Systems Platforms, volume 1795 of Lecture Notes in Computer Science, pp. 71-87, Springer, Apr. 2000.

[Rac01] Rackl, G., Monitoring and Managing Heterogeneous Middleware, volume 23 of LRR-TUM Research Report Series. Shaker Verlag, Aachen, 2001.

[Raj92] Raju, S.C.V., Rajkumar, R., Jahanian, F.,Timing Constraints Monitoring in Distributed Real-Time Systems, Proc of 13th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, pp.57-67, December 1992.

[Sch00] Schmidt, D. C., Kuhns, F., An Overview of the Real-Time CORBA Specification, IEEE Computer Magazine, Special Issue on Object-oriented Real-time Computing, Vol. 33, No. 6, pp. 56-63, 2000.

[Sch97] Schmidt, D., R. Bector, D. Levine, S. Mungee, G. Parulkar, TAO: A Middleware Framework for Real-time ORB Endsystems, Proc. 1997 IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, San Francisco, CA, 1997.

[Sel94] Selic, Bran et al., Real-Time Object-Oriented Modeling, John Wiley & Sons, New York, USA, 1994.

[Sho99] Shokri, E., Kim, K., TMO-Based Programming in COTS Software/Hardware Platforms: A Case Study, Proc. ASSET '99 (1999 IEEE Symp. on Application-Specific Systems and Software Engineering & Technology), Richardson, TX, pp. 88-94, 1999.

[Sta89] Stankovic, J.A. and K. Ramamritham, The Spring Kernel: A New Paradigm for Real-Time Operating Systems, ACM Operating Systems Review, Vol. 23 No.3, July 1989.

[Str95] Streich, H., TaskPair-Scheduling: An Approach for Dynamic Real-Time Systems, Int. Journal of Mini & Microcomputers, Vol. 17, No. 2, pp 77-83, 1995.

[Tok88] Tokuda, H., Kotera, M., Mercer, C. W., A real-time monitor for a distributed real-time operating system, Proc. of ACM Workshop on Parallel and Distributed Debugging, Madison, WI, USA, pp. 68-77, May 1988.

[Tsa96] Tsai, J., Bi, Y., Yang, S., Smith, R., Distributed Real-Time Systems - Monitoring, Visualization, Debugging, and Analysis, John Wiley & Sons, New York, USA, 1996.